

modeled the layers of skin accounting for just a single scattering event in each layer, and Dorsey and Hanrahan (1996) rendered layered materials using the Kubelka–Munk theory, which accounts for multiple scattering within layers but assumes that radiance distribution doesn't vary as a function of direction.

Pharr and Hanrahan (2000) showed that Monte Carlo integration could be used to solve the *adding equations* to efficiently compute BSDFs for layered materials without needing either of these simplifications. The adding equations are integral equations that accurately describe the effect of multiple scattering in layered media that were derived by van De Hulst (1980) and Twomey et al. (1966). Weidlich and Wilkie (2007) rendered layered materials more efficiently by making a number of simplifying assumptions, and Jakob et al. (2014a) efficiently computed scattering in layered materials using the Fourier basis representation implemented here as the `FourierBSDF`.

A number of researchers have investigated BRDFs based on modeling the small-scale geometric features of a reflective surface. This work includes the computation of BRDFs from bump maps by Cabral, Max, and Springmeyer (1987), Fournier's normal distribution functions (Fournier 1992), and Westin, Arvo, and Torrance (1992), who applied Monte Carlo ray tracing to statistically model reflection from microgeometry and represented the resulting BRDFs with spherical harmonics. More recently, Wu et al. (2011) developed a system that made it possible to model microgeometry and specify its underlying BRDF while interactively previewing the resulting macro-scale BRDF.

Improvements in data-acquisition technology have led to increasing amounts of detailed real-world BRDF data, even including BRDFs that are spatially varying (sometimes called “bidirectional texture functions,” BTFs) (Dana et al. 1999). Matusik et al. (2003a, 2003b) assembled an early database of measured isotropic BRDF data. See Müller et al. (2005) for a survey of work in BRDF measurement until the year 2005. Sun et al. (2007) measured BRDFs as they change over time—for example, due to paint drying, a wet surface becoming dry, or dust accumulating. While most BRDF measurement has been based on measuring reflected radiance due to a given amount of incident irradiance, Zhao et al. (2011) showed that CT imaging of the structure of fabrics led to very accurate reflection models.

Fitting measured BRDF data to parametric reflection models is a difficult problem. Rusinkiewicz (1998) made the influential observation that reparameterizing the measured data can make it substantially easier to compress or fit to models; this topic has been further investigated by Stark et al. (2005) and in Marscher's Ph.D. dissertation (1998). Ngan et al. (2005) analyzed the effectiveness of a variety of BRDF models for fitting measured data and showed that models based on the half-angle vector, rather than a reflection vector, tended to be more effective. See also the paper on this topic by Edwards et al. (2005).

Zickler et al. (2005) developed a method for representing BRDFs based on radial basis functions (RBFs)—they use them to interpolate irregularly sampled 5D spatially varying BRDFs. Weistroffer et al. (2007) have shown how to efficiently represent scattered reflectance data with RBFs without needing to resample them to have regular spacing. Wang et al. (2008a) demonstrated a successful approach for acquiring spatially varying anisotropic BRDFs. Pacanowski et al. (2012) developed a representation that could

guarantee a given error bound between measured and fit data, and Bagher et al. (2012) introduced a parametric BRDF that accurately fit a wide variety of reflection function distributions using just six coefficients per color channel. More recently, Brady et al. (2014) found new analytic BRDF models that fit measured BRDFs well using genetic programming. Dupuy et al. (2015) developed an efficient and easily implemented approach for fitting measured BRDFs to microfacet distributions based on using power iterations to compute eigenvectors.

Kajiya and Kay (1989) developed an early reflection model for hair based on a model of individual hairs as cylinders with diffuse and glossy reflection properties. Their model determines the overall reflection from these cylinders, accounting for the effect of variation in surface normal over the hemisphere along the cylinder. For related work, see also the paper by Banks (1994), which discusses shading models for 1D primitives like hair. Goldman (1997) developed a probabilistic shading model that models reflection from collections of short hairs. Marschner et al. (2003) developed an accurate model of light scattering from human hair fibers that decomposes the reflected light into three components with distinct directional profiles based on the number of internal refraction events. Sadeghi et al. (2010) developed intuitive controls for physically based hair reflection models that made it easier to achieve a desired visual appearance. Further improvements to hair scattering models were introduced by d’Eon et al. (2011). Finally, see Ward et al.’s survey (2007) for extensive coverage of research in modeling, animating, and rendering hair.

Modeling reflection from a variety of specific types of surfaces has received attention from researchers, leading to specialized reflection models. Examples include Marschner et al.’s (2005) work on rendering wood, Günther et al.’s (2005) investigation of car paint, and Papas et al.’s (2014) model for paper.

Cloth remains particularly challenging material to render. Work in this area includes papers by Sattler et al. (2003), Irawan (2008), Schröder et al. (2011), Irawan and Marschner (2012), Zhao et al. (2012), and Sadeghi et al. (2013).

Nayar, Ikeuchi, and Kanade (1991) have shown that some reflection models based on physical (wave) optics have substantially similar characteristics to those based on geometric optics. The geometric optics approximations don’t seem to cause too much error in practice, except on very smooth surfaces. This is a helpful result, giving experimental basis to the general belief that wave optics models aren’t usually worth their computational expense for computer graphics applications.

The effect of the polarization of light is not modeled in `pbrt`, although for some scenes it can be an important effect; see, for example, the paper by Tannenbaum, Tannenbaum, and Wozny (1994) for information about how to extend a renderer to account for this effect. Similarly, the fact that indices of refraction of real-world objects usually vary as a function of wavelength is also not modeled here; see both Section 11.8 of Glassner’s book (1995) and Devlin et al.’s survey article for information about these issues and references to previous work (Devlin et al. 2002). Fluorescence, where light is reflected at different wavelengths than the incident illumination, is also not modeled by `pbrt`; see Glassner (1994) and Wilkie et al. (2006) for more information on this topic.

Moravec (1981) was the first to apply a wave optics model to graphics. This area has also been investigated by Bahar and Chakrabarti (1987) and Stam (1999), who applied wave optics to model diffraction effects. For more recent work in this area, see the papers by Cuypers et al. (2012) and Musbach et al. (2013), who also provide extensive references to previous work on this topic.

EXERCISES

- 8.1** A consequence of Fermat's principle from optics is that light traveling from a point p_1 in a medium with index of refraction η_1 to a point p_2 in a medium with index of refraction η_2 will follow a path that minimizes the time to get from the first point to the second point. Snell's law can be shown to follow from this fact directly.

Consider light traveling between two points p_1 and p_2 separated by a planar boundary. The light could potentially pass through the boundary while traveling from p_1 to p_2 at any point on the boundary (see Figure 8.25, which shows two such possible points p' and p''). Recall that the time it takes light to travel between two points in a medium with a constant index of refraction is proportional to the distance between them times the index of refraction in the medium. Using this fact, show that the point p' on the boundary that minimizes the total time to travel from p_1 to p_2 is the point where $\eta_1 \sin \theta_1 = \eta_2 \sin \theta_2$.

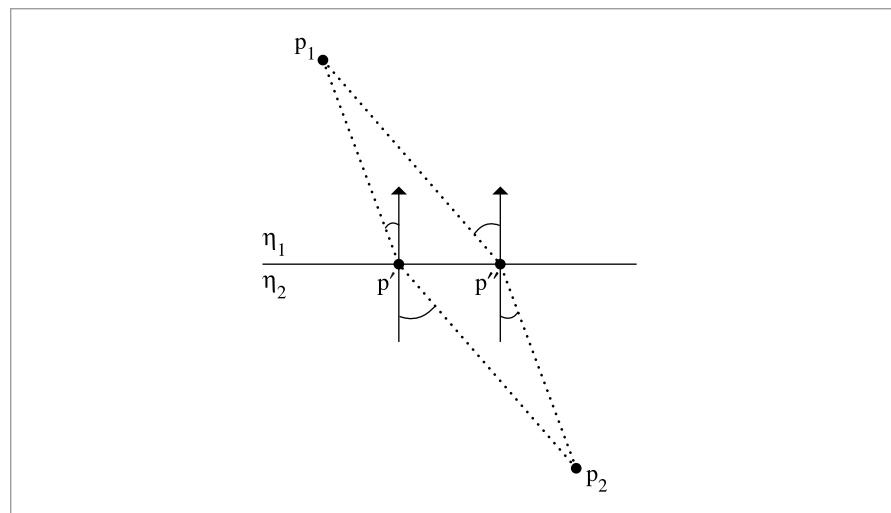


Figure 8.25: Derivation of Snell's Law. Snell's law can be derived using Fermat's principle, which says that light will follow the path that takes the least amount of time to pass between two points. The angle of refraction θ at the boundary between two media can thus be shown to be the one that minimizes the time spent going from p_1 to a point p on the boundary plus the time spent traveling the distance from that point to p_2 .

- ③ 8.2 Read the papers of Wolff and Kurlander (1990) and Tannenbaum, Tannenbaum, and Wozny (1994), and apply some of the techniques described to modify pbrt to model the effect of light polarization. Set up scenes and render images of them that demonstrate a significant difference when polarization is accurately modeled.
- ③ 8.3 Construct a scene with an actual geometric model of a rough plane with a large number of mirrored microfacets, and illuminate it with an area light source.¹¹ Place the camera in the scene such that a very large number of microfacets are in each pixel's area, and render images of this scene using hundreds or thousands of pixel samples. Compare the result to using a flat surface with a microfacet-based BRDF model. How well can you get the two approaches to match if you try to tune the microfacet BRDF parameters? Can you construct examples where images rendered with the true microfacets are actually visibly more realistic due to better modeling the effects of masking, self-shadowing, and interreflection between microfacets?
- ③ 8.4 Extend pbrt to be able to more accurately render interesting surfaces like wood (Marschner et al. 2005), cloth (Sattler et al. 2003), or car paint (Günther et al. 2005). Render images showing better visual results than when existing reflection functions in pbrt are used for these effects.
- ③ 8.5 Implement a simulation-based approach to modeling reflection from complex microsurfaces, such as the one described by Westin, Arvo, and Torrance (1992). Modify pbrt so that you can provide a description of the microgeometry of a complex surface (like cloth, velvet, etc.), fire rays at the geometry from a variety of incident directions, and record the distribution and throughput for the rays that leave the surface. (You will likely need to modify the PathIntegrator from Chapter 14 to determine the distribution of outgoing light.) Record the distribution in a 3D table if the surface is isotropic or a 4D table if it is anisotropic, and use the table to compute BRDF values for rendering images. Demonstrate interesting reflection effects from complex surfaces using this approach. Investigate how the size of the table and the number of samples taken to compute entries in the table affect the accuracy of the final result.
- ② 8.6 Although pbrt features a Curve shape that provides fairly efficient intersection tests between rays and parametric curves (Section 3.7), it lacks a reflection model for hair. Choose one of the models described in the “Further Reading” section such as Marschner et al.’s (2003) or d’Eon et al.’s (2011), and implement it in pbrt. Either find a geometric model of hair or generate hair procedurally, and render images using your implementation.

[Curve 168](#)

[PathIntegrator 875](#)

¹¹ An area light and not a point or directional light is necessary due to subtleties in how lights are seen in specular surfaces. With the light transport algorithms used in pbrt, infinitesimal point sources are never visible in mirrored surfaces. This is a typical limitation of ray-tracing renderers and usually not bothersome in practice.

This page intentionally left blank

CHAPTER NINE



09 MATERIALS

The BRDFs and BTDFs introduced in the previous chapter address only part of the problem of describing how a surface scatters light. Although they describe how light is scattered at a particular point on a surface, the renderer needs to determine *which* BRDFs and BTDFs are present at a point on a surface and what their parameters are. In this chapter, we describe a procedural shading mechanism that addresses this issue.

The basic idea behind *pbrt*'s approach is that a *surface shader* is bound to each primitive in the scene. The surface shader is represented by an instance of the `Material` interface class, which has a method that takes a point on a surface and creates a `BSDF` object (and possibly a `BSSRDF`) that represents scattering at the point. The `BSDF` class holds a set of `BxDFs` whose contributions are summed to give the full scattering function. Materials, in turn, use instances of the `Texture` class (to be defined in the next chapter) to determine the material properties at particular points on surfaces. For example, an `ImageTexture` might be used to modulate the color of diffuse reflection across a surface. This is a somewhat different shading paradigm from the one that many rendering systems use; it is common practice to combine the function of the surface shader and the lighting integrator (see Chapter 14) into a single module and have the shader return the color of reflected light at the point. However, by separating these two components and having the `Material` return a `BSDF`, *pbrt* is better able to handle a variety of light transport algorithms.

9.1 BSDFs

The `BSDF` class represents a collection of BRDFs and BTDFs. Grouping them in this manner allows the rest of the system to work with composite `BSDFs` directly, rather than having to consider all of the components they may have been built from. Equally important, the

BSDF class hides some of the details of shading normals from the rest of the system. Shading normals, either from per-vertex normals in triangle meshes or from bump mapping, can substantially improve the visual richness of rendered scenes, but because they are an *ad hoc* construct, they are tricky to incorporate into a physically based renderer. The issues that they introduce are handled in the BSDF implementation.

(BSDF Declarations) +≡

```
class BSDF {
public:
    (BSDF Public Methods 573)
    (BSDF Public Data 573)
private:
    (BSDF Private Methods 576)
    (BSDF Private Data 573)
};
```

The BSDF constructor takes a SurfaceInteraction object that contains information about the differential geometry at the point on a surface as well as a parameter eta that gives the relative index of refraction over the boundary. For opaque surfaces, eta isn't used, and a value of one should be provided by the caller. (The default value of one for eta is for just this case.) The constructor computes an orthonormal coordinate system with the shading normal as one of the axes; this coordinate system will be useful for transforming directions to and from the BxDF coordinate system that is described in Figure 8.2. Throughout this section, we will use the convention that n_s denotes the shading normal and n_g the geometric normal (Figure 9.1).

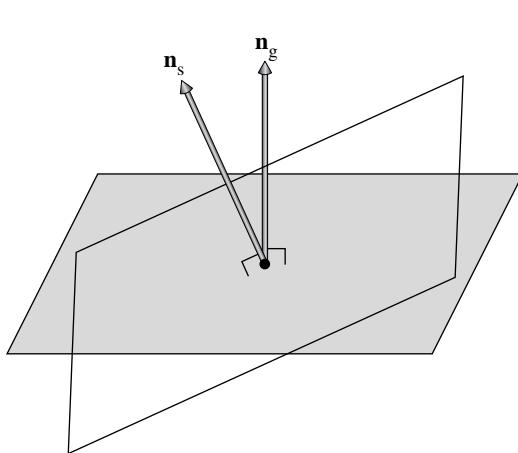


Figure 9.1: The geometric normal, n_g , defined by the surface geometry, and the shading normal, n_s , given by per-vertex normals and/or bump mapping, will generally define different hemispheres for integrating incident illumination to compute surface reflection. This inconsistency is important to handle carefully since it can otherwise lead to artifacts in images.

BSDF 572

BxDF 513

SurfaceInteraction 116

{BSDF Public Methods} ≡ 572
 BSDF(const SurfaceInteraction &si, Float eta = 1)
 : eta(eta), ns(si.shading.n), ng(si.n),
 ss(Normalize(si.shading.dpdu)), ts(Cross(ns, ss)) { }

{BSDF Public Data} ≡ 572
 const Float eta;

{BSDF Private Data} ≡ 572
 const Normal3f ns, ng;
 const Vector3f ss, ts;

The BSDF implementation stores only a limited number of individual BxDF components. It could easily be extended to allocate more space if more components were given to it, although this isn't necessary for any of the Material implementations in pbrt thus far, and the current limit of eight is plenty for almost all practical applications.

{BSDF Public Methods} +≡ 572
 void Add(BxDF *b) {
 Assert(nBxDFs < MaxBxDFs);
 bxfds[nBxDFs++] = b;
}

{BSDF Private Data} +≡ 572
 int nBxDFs = 0;
 static constexpr int MaxBxDFs = 8;
 BxDF *bxfds[MaxBxDFs];

For other parts of the system that need additional information about the particular BRDFs and BTDFs that are present, a method returns the number of BxDFs stored by the BSDF that match a particular set of BxDFType flags.

{BSDF Public Methods} +≡ 572
 int NumComponents(BxDFType flags = BSDF_ALL) const;

Assert() 1069
 BSDF 572
 BSDF::bxfds 573
 BSDF::nBxDFs 573
 BSDF_ALL 513
 BxDF 513
 BxDFType 513
 Cross() 65
 Float 1062
 Material 577
 Normal3f 71
 SurfaceInteraction 116
 Vector3f::Normalize() 66
 Vector3f 60

The BSDF also has methods that perform transformations to and from the local coordinate system used by BxDFs. Recall that, in this coordinate system, the surface normal is along the z axis $(0, 0, 1)$, the primary tangent is $(1, 0, 0)$, and the secondary tangent is $(0, 1, 0)$. The transformation of directions into “shading space” simplifies many of the BxDF implementations in Chapter 8. Given three orthonormal vectors s , t , and n in world space, the matrix M that transforms vectors in world space to the local reflection space is

$$M = \begin{pmatrix} s_x & s_y & s_z \\ t_x & t_y & t_z \\ n_x & n_y & n_z \end{pmatrix} = \begin{pmatrix} s \\ t \\ n \end{pmatrix}.$$

To confirm this yourself, consider, for example, the value of M times the surface normal n , $Mn = (s \cdot n, t \cdot n, n \cdot n)$. Since s , t , and n are all orthonormal, the x and y components of Mn are zero. Since n is normalized, $n \cdot n = 1$. Thus, $Mn = (0, 0, 1)$, as expected.

In this case, we don't need to compute the inverse transpose of M to transform normals (recall the discussion of transforming normals in Section 2.8.3). Because M is an orthogonal matrix (its rows and columns are mutually orthogonal), its inverse is equal to its transpose, so it is its own inverse transpose already.

(BSDF Public Methods) +≡

572

```
Vector3f WorldToLocal(const Vector3f &v) const {
    return Vector3f(Dot(v, ss), Dot(v, ts), Dot(v, ns));
}
```

The method that takes vectors back from local space to world space transposes M to find its inverse before doing the appropriate dot products.

(BSDF Public Methods) +≡

572

```
Vector3f LocalToWorld(const Vector3f &v) const {
    return Vector3f(ss.x * v.x + ts.x * v.y + ns.x * v.z,
                    ss.y * v.x + ts.y * v.y + ns.y * v.z,
                    ss.z * v.x + ts.z * v.y + ns.z * v.z);
}
```

Shading normals can cause a variety of undesirable artifacts in practice (Figure 9.2). Figure 9.2(a) shows a *light leak*: the geometric normal indicates that ω_i and ω_o lie on opposite sides of the surface, so if the surface is not transmissive, the light should have no contribution. However, if we directly evaluate the scattering equation, Equation (5.9), about the hemisphere centered around the shading normal, we will incorrectly incorporate the light from ω_i . This case demonstrates that n_s can't just be used as a direct replacement for n_g in rendering computations.

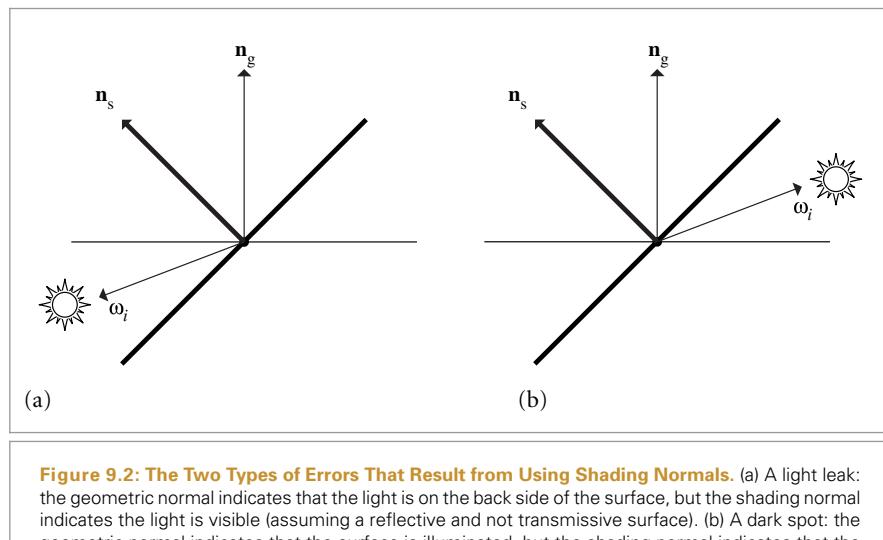


Figure 9.2: The Two Types of Errors That Result from Using Shading Normals. (a) A light leak: the geometric normal indicates that the light is on the back side of the surface, but the shading normal indicates the light is visible (assuming a reflective and not transmissive surface). (b) A dark spot: the geometric normal indicates that the surface is illuminated, but the shading normal indicates that the viewer is behind the lit side of the surface.

BSDF::ns 573

BSDF::ss 573

BSDF::ts 573

Dot() 63

Vector3f 60

Figure 9.2(b) shows a similar tricky situation: the shading normal indicates that no light should be reflected to the viewer, since it is not in the same hemisphere as the illumination, while the geometric normal indicates that they are in the same hemisphere. Direct use of \mathbf{n}_s would cause ugly black spots on the surface where this situation happens.

Fortunately, there is an elegant solution to these problems. When evaluating the BSDF, we can use the geometric normal to decide if we should be evaluating reflection or transmission: if ω_i and ω_o lie in the same hemisphere with respect to \mathbf{n}_g , we evaluate the BRDFs, and otherwise we evaluate the BTDFs. In evaluating the scattering equation, however, the dot product of the normal and the incident direction is still taken with the shading normal rather than the geometric normal.

Now it should be clear why pbrt requires BxDFs to evaluate their values without regard to whether ω_i and ω_o are in the same or different hemispheres. This convention means that light leaks are avoided, since we will only evaluate the BTDFs for the situation in Figure 9.2(a), giving no reflection for a purely reflective surface. Similarly, black spots are avoided since we will evaluate the BRDFs for the situation in Figure 9.2(b), even though the shading normal would suggest that the directions are in different hemispheres.

Given these conventions, the method that evaluates the BSDF for a given pair of directions follows directly. It starts by transforming the world space direction vectors to local BSDF space and then determines whether it should use the BRDFs or the BTDFs. It then loops over the appropriate set and evaluates the sum of their contributions.

{BSDF Method Definitions} ≡

```
Spectrum BSDF::f(const Vector3f &woW, const Vector3f &wiW,
                  BxDFType flags) const {
    Vector3f wi = WorldToLocal(wiW), wo = WorldToLocal(woW);
    bool reflect = Dot(wiW, ng) * Dot(woW, ng) > 0;
    Spectrum f(0.f);
    for (int i = 0; i < nBxDFs; ++i)
        if (bxdfs[i]->MatchesFlags(flags) &&
            ((reflect && (bxdfs[i]->type & BSDF_REFLECTION)) ||
             (!reflect && (bxdfs[i]->type & BSDF_TRANSMISSION))))
            f += bxdfs[i]->f(wo, wi);
}
```

BSDF 572

BSDF::bxdfs 573

BSDF::nBxDFs 573

BSDF::WorldToLocal() 574

BxDF_ALL 513

BSDF_REFLECTION 513

BSDF_TRANSMISSION 513

BxDF 513

BxDF::f() 514

BxDF::MatchesFlags() 513

BxDF::rho() 515

BxDFType 513

Dot() 63

Point2f 68

Spectrum 315

Vector3f 60

pbrt also provides BSDF methods that return the BSDF's reflectances. (Recall the definition of reflectance in Section 8.1.1.) The two corresponding methods just loop over the BxDFs and sum the values returned by their BxDF::rho() methods; their straightforward implementations aren't included here. These methods take arrays of samples for BxDFs for use in Monte Carlo sampling algorithms if needed (recall the BxDF::rho() interface defined in Section 8.1.1, which takes such samples as well.)

{BSDF Public Methods} +≡

572

```
Spectrum rho(int nSamples, const Point2f *samples1,
              const Point2f *samples2, BxDFType flags = BSDF_ALL) const;
Spectrum rho(const Vector3f &wo, int nSamples, const Point2f *samples,
              BxDFType flags = BSDF_ALL) const;
```

9.1.1 BSDF MEMORY MANAGEMENT

For each ray that intersects geometry in the scene, one or more BSDF objects will be created by the Integrator in the process of computing the radiance carried along the ray. (Integrators that account for multiple interreflections of light will generally create a number of BSDFs along the way.) Each of these BSDFs in turn has a number of BxDFs stored inside it, as created by the Materials at the intersection points.

A naïve implementation would use new and delete to dynamically allocate storage for both the BSDF as well as each of the BxDFs that it holds. Unfortunately, such an approach would be unacceptably inefficient—too much time would be spent in the dynamic memory management routines for a series of small memory allocations. Instead, the implementation here uses a specialized allocation scheme based on the MemoryArena class described in Section A.4.3.¹ A MemoryArena is passed into methods that allocate memory for BSDFs. For example, the SamplerIntegrator::Render() method creates a MemoryArena for each image tile and passes it to the integrators, which in turn pass it to the Material.

For the convenience of code that allocates BSDFs and BxDFs (e.g., the Materials in this chapter), there is a macro that hides some of the messiness of using the memory arena. Instead of using the new operator to allocate those objects like this:

```
BSDF *b = new BSDF;
BxDF *lam = new LambertianReflection(Spectrum(0.5f));
```

code should instead be written with the ARENA_ALLOC() macro, like this:

```
BSDF *b = ARENA_ALLOC(arena, BSDF);
BxDF *lam = ARENA_ALLOC(arena, LambertianReflection)(Spectrum(0.5f));
```

where arena is a MemoryArena.

The ARENA_ALLOC() macro uses the placement operator new to run the constructor for the object at the returned memory location.

(Memory Declarations) ≡

```
#define ARENA_ALLOC(arena, Type) new (arena.Alloc(sizeof(Type))) Type
```

The BSDF destructor is a private method in order to ensure that it isn't inadvertently called (e.g., due to an attempt to delete a BSDF). Making the destructor private ensures a compile time error if it is called. Trying to delete memory allocated by the MemoryArena could lead to errors or crashes, since a pointer to the middle of memory managed by the MemoryArena would be passed to the system's dynamic memory freeing routine.

In turn, an implication of the allocation scheme here is that BSDF and BxDF destructors are never executed. This isn't a problem for the ones currently implemented in the system.

(BSDF Private Methods) ≡

```
-BSDF() { }
```

ARENA_ALLOC()	576
BSDF	572
BxDF	513
Integrator	25
Material	577
MemoryArena	1074
MemoryArena::Alloc()	1074
MemoryArena::Reset()	1076
SamplerIntegrator::Render()	26

¹ MemoryArena allocates a large block of memory and responds to allocation requests via the MemoryArena::Alloc() call by returning successive sections of that block. It does not support freeing individual allocations but instead frees all of them simultaneously when the MemoryArena::Reset() method is called. The result of this approach is that both allocation and freeing of memory are extremely efficient.

9.2 MATERIAL INTERFACE AND IMPLEMENTATIONS

The abstract `Material` class defines the interface that material implementations must provide. The `Material` class is defined in the files `core/material.h` and `core/material.cpp`.

```
(Material Declarations) ≡
class Material {
public:
    (Material Interface 577)
};
```

A single method must be implemented by Materials: `ComputeScatteringFunctions()`. This method is given a `SurfaceInteraction` object that contains geometric properties at an intersection point on the surface of a shape. The method's implementation is responsible for determining the reflective properties at the point and initializing the `SurfaceInteraction::bsdf` member variable with a corresponding BSDF class instance. If the material includes subsurface scattering, then the `SurfaceInteraction::bssrdf` member should be initialized as well. (It should otherwise be left unchanged from its default `nullptr` value.) The `BSSRDF` class that represents subsurface scattering functions is defined later, in Section 11.4, after the foundations of volumetric scattering have been introduced.

Three additional parameters are passed to this method:

- A `MemoryArena`, which should be used to allocate memory for BSDFs and BSSRDFs.
- The `TransportMode` parameter, which indicates whether the surface intersection was found along a path starting from the camera or one starting from a light source; this detail has implications for how BSDFs and BSSRDFs are evaluated—see Section 16.1.
- Finally, the `allowMultipleLobes` parameter indicates whether the material should use BxDFs that aggregate multiple types of scattering into a single BxDF when such BxDFs are available. (An example of such a BxDF is `FresnelSpecular`, which includes both specular reflection and transmission.) These BxDFs can improve the quality of final results when used with Monte Carlo light transport algorithms but can introduce noise in images when used with the `DirectLightingIntegrator` and `WhittedIntegrator`. Therefore, the Integrator is allowed to control whether such BxDFs are used via this parameter.

(Material Interface) ≡

```
virtual void ComputeScatteringFunctions(SurfaceInteraction *si,
                                         MemoryArena &arena, TransportMode mode,
                                         bool allowMultipleLobes) const = 0;
```

Since the usual interface to the intersection point used by Integrators is through an instance of the `SurfaceInteraction` class, we will add a convenience method `ComputeScatteringFunctions()` to that class. Its implementation first calls the `SurfaceInteraction`'s `ComputeDifferentials()` method to compute information about the projected size of the surface area around the intersection on the image plane for use in texture antialiasing. Next, it forwards the request to the `Primitive`, which in turn will

BSDF 572
BSSRDF 692
BxDF 513
DirectLightingIntegrator 851
FresnelSpecular 531
Integrator 25
Material 577
MemoryArena 1074
Primitive 248
SurfaceInteraction 116
SurfaceInteraction::bsdf 250
SurfaceInteraction::bssrdf 250
TransportMode 960
WhittedIntegrator 32

call the corresponding `ComputeScatteringFunctions()` method of its `Material`. (See, for example, the `GeometricPrimitive::ComputeScatteringFunctions()` implementation.)

```
(SurfaceInteraction Method Definitions) +≡
void SurfaceInteraction::ComputeScatteringFunctions(
    const RayDifferential &ray, MemoryArena &arena,
    bool allowMultipleLobes, TransportMode mode) {
    ComputeDifferentials(ray);
    primitive->ComputeScatteringFunctions(this, arena, mode,
        allowMultipleLobes);
}
```

9.2.1 MATTE MATERIAL

The `MatteMaterial` material is defined in `materials/matte.h` and `materials/matte.cpp`. It is the simplest material in `pbrt` and describes a purely diffuse surface.

```
(MatteMaterial Declarations) ≡
class MatteMaterial : public Material {
public:
    (MatteMaterial Public Methods 578)
private:
    (MatteMaterial Private Data 578)
};
```

This material is parameterized by a spectral diffuse reflection value, `Kd`, and a scalar roughness value, `sigma`. If `sigma` has the value zero at the point on a surface, `MatteMaterial` creates a `LambertianReflection` BRDF; otherwise, the `OrenNayar` model is used. Like all of the other `Material` implementations in this chapter, it also takes an optional scalar texture that defines an offset function over the surface. If its value is not `nullptr`, this texture is used to compute a shading normal at each point based on the function it defines. (Section 9.3 discusses the implementation of this computation.) Figure 8.14 in the previous chapter shows the `MatteMaterial` material with the dragon model.

```
(MatteMaterial Public Methods) ≡ 578
MatteMaterial(const std::shared_ptr<Texture<Spectrum>> &Kd,
              const std::shared_ptr<Texture<Float>> &sigma,
              const std::shared_ptr<Texture<Float>> &bumpMap)
: Kd(Kd), sigma(sigma), bumpMap(bumpMap) { }
```



```
(MatteMaterial Private Data) ≡ 578
std::shared_ptr<Texture<Spectrum>> Kd;
std::shared_ptr<Texture<Float>> sigma, bumpMap;
```

The `ComputeScatteringFunctions()` method puts the pieces together, determining the bump map's effect on the shading geometry, evaluating the textures, and allocating and returning the appropriate BSDF.

BSDF 572
 Float 1062
`GeometricPrimitive::ComputeScatteringFunctions()` 251
`LambertianReflection` 532
`Material` 577
`MatteMaterial` 578
`MatteMaterial::bumpMap` 578
`MatteMaterial::Kd` 578
`MatteMaterial::sigma` 578
`MemoryArena` 1074
`OrenNayar` 536
`Primitive::ComputeScatteringFunctions()` 250
`RayDifferential` 75
`Spectrum` 315
`SurfaceInteraction` 116
`SurfaceInteraction::ComputeDifferentials()` 601
`Texture` 614
`TransportMode` 960

```
<MatteMaterial Method Definitions> ≡
    void MatteMaterial::ComputeScatteringFunctions(SurfaceInteraction *si,
        MemoryArena &arena, TransportMode mode,
        bool allowMultipleLobes) const {
    <Perform bump mapping with bumpMap, if present 579>
    <Evaluate textures for MatteMaterial material and allocate BRDF 579>
}
```

If a bump map was provided to the `MatteMaterial` constructor, the `Material::Bump()` method is called to calculate the shading normal at the point. This method will be defined in the next section.

```
<Perform bump mapping with bumpMap, if present> ≡
    if (bumpMap)
        Bump(bumpMap, si);
```

579, 581, 584, 701

Next, the `Textures` that give the values of the diffuse reflection spectrum and the roughness are evaluated; texture implementations may return constant values, look up values from image maps, or do complex procedural shading calculations to compute these values (the texture evaluation process is the subject of Chapter 10). Given these values, all that needs to be done is to allocate a `BSDF` and then allocate the appropriate type of Lambertian `BRDF` and provide it to the `BSDF`. Because `Textures` may return negative values or values otherwise outside of the expected range, these values are clamped to valid ranges before they are passed to the `BRDF` constructor.

```
<Evaluate textures for MatteMaterial material and allocate BRDF> ≡
    si->bsdf = ARENA_ALLOC(arena, BSDF)(*si);
    Spectrum r = Kd->Evaluate(*si).Clamp();
    Float sig = Clamp(sigma->Evaluate(*si), 0, 90);
    if (!r.IsBlack()) {
        if (sig == 0)
            si->bsdf->Add(ARENA_ALLOC(arena, LambertianReflection)(r));
        else
            si->bsdf->Add(ARENA_ALLOC(arena, OrenNayar)(r, sig));
    }
```

579

ARENA_ALLOC() 576
`BSDF` 572
`BSDF::Add()` 573
`Float` 1062
`LambertianReflection` 532
`Material::Bump()` 589
`MatteMaterial` 578
`MatteMaterial::bumpMap` 578
`MatteMaterial::Kd` 578
`MatteMaterial::sigma` 578
`MemoryArena` 1074
`OrenNayar` 536
`PlasticMaterial` 580
`Spectrum` 315
`Spectrum::Clamp()` 317
`Spectrum::IsBlack()` 317
`SurfaceInteraction` 116
`SurfaceInteraction::bsdf` 250
`Texture` 614
`Texture::Evaluate()` 615
`TransportMode` 960

9.2.2 PLASTIC MATERIAL

Plastic can be modeled as a mixture of a diffuse and glossy scattering function with parameters controlling the particular colors and specular highlight size. The parameters to `PlasticMaterial` are two reflectivities, `Kd` and `Ks`, which respectively control the amounts of diffuse reflection and glossy specular reflection.

Next is a roughness parameter that determines the size of the specular highlight. It can be specified in two ways. First, if the `remapRoughness` parameter is `true`, then the given roughness should vary from zero to one, where the higher the roughness value, the larger the highlight. (This variant is intended to be fairly user-friendly.) Alternatively, if the parameter is `false`, then the roughness is used to directly initialize the microfacet distribution's α parameter (recall Section 8.4.2).



Figure 9.3: Dragon Rendered with a Plastic Material. Note the combination of diffuse and glossy specular reflection. (Model courtesy of Christian Schüller.)

Figure 9.3 shows a plastic dragon. `PlasticMaterial` is defined in `materials/plastic.h` and `materials/plastic.cpp`.

```
(PlasticMaterial Declarations) ≡
class PlasticMaterial : public Material {
public:
    (PlasticMaterial Public Methods 580)
private:
    (PlasticMaterial Private Data 580)
};

(PlasticMaterial Public Methods) ≡
580 PlasticMaterial(const std::shared_ptr<Texture<Spectrum>> &Kd,
                     const std::shared_ptr<Texture<Spectrum>> &Ks,
                     const std::shared_ptr<Texture<Float>> &roughness,
                     const std::shared_ptr<Texture<Float>> &bumpMap,
                     bool remapRoughness)
    : Kd(Kd), Ks(Ks), roughness(roughness), bumpMap(bumpMap),
      remapRoughness(remapRoughness) { }

(PlasticMaterial Private Data) ≡
580 std::shared_ptr<Texture<Spectrum>> Kd, Ks;
std::shared_ptr<Texture<Float>> roughness, bumpMap;
const bool remapRoughness;
```

Float 1062
Material 577
PlasticMaterial 580
PlasticMaterial::bumpMap 580
PlasticMaterial::Kd 580
PlasticMaterial::Ks 580
PlasticMaterial::
 remapRoughness
 580
PlasticMaterial::roughness
 580
Spectrum 315
Texture 614

The `PlasticMaterial::ComputeScatteringFunctions()` method follows the same basic structure as `MatteMaterial::ComputeScatteringFunctions()`: it calls the bump-mapping function, evaluates textures, and then allocates BxDFs to use to initialize the BSDF.

(PlasticMaterial Method Definitions) ≡

```
void PlasticMaterial::ComputeScatteringFunctions(
    SurfaceInteraction *si, MemoryArena &arena, TransportMode mode,
    bool allowMultipleLobes) const {
    (Perform bump mapping with bumpMap, if present 579)
    si->bsdf = ARENA_ALLOC(arena, BSDF)(*si);
    (Initialize diffuse component of plastic material 581)
    (Initialize specular component of plastic material 581)
}
```

ARENA_ALLOC() 576
 BSDF 572
 BSDF::Add() 573
 BxDF 513
 Float 1062
 Fresnel 521
 FresnelDielectric 522
 LambertianReflection 532
 Material 577
 MatteMaterial::
 ComputeScatteringFunctions() 579
 MemoryArena 1074
 MicrofacetDistribution 537
 MicrofacetReflection 547
 MiMaterial 582
 PlasticMaterial::
 ComputeScatteringFunctions() 581
 PlasticMaterial::Kd 580
 PlasticMaterial::Ks 580
 PlasticMaterial::
 remapRoughness 580
 PlasticMaterial::roughness 580
 Spectrum 315
 Spectrum::Clamp() 317
 Spectrum::IsBlack() 317
 SurfaceInteraction 116
 SurfaceInteraction::bsdf 250
 Texture::Evaluate() 615
 TransportMode 960
 TrowbridgeReitzDistribution 540
 TrowbridgeReitzDistribution::
 RoughnessToAlpha() 540

In Material implementations, it's worthwhile to skip creation of BxDF components that do not contribute to the scattering at a point. Doing so saves the renderer unnecessary work later when it's computing reflected radiance at the point. Therefore, the Lambertian component is only created if `kd` is non-zero.

(Initialize diffuse component of plastic material) ≡

```
Spectrum kd = Kd->Evaluate(*si).Clamp();
if (!kd.IsBlack())
    si->bsdf->Add(ARENA_ALLOC(arena, LambertianReflection)(kd));
```

As with the diffuse component, the glossy specular component is skipped if it's not going to make a contribution to the overall BSDF.

(Initialize specular component of plastic material) ≡

```
Spectrum ks = Ks->Evaluate(*si).Clamp();
if (!ks.IsBlack()) {
    Fresnel *fresnel = ARENA_ALLOC(arena, FresnelDielectric)(1.f, 1.5f);
    (Create microfacet distribution distrib for plastic material 581)
    BxDF *spec =
        ARENA_ALLOC(arena, MicrofacetReflection)(ks, distrib, fresnel);
    si->bsdf->Add(spec);
}
```

(Create microfacet distribution distrib for plastic material) ≡

```
Float rough = roughness->Evaluate(*si);
if (remapRoughness)
    rough = TrowbridgeReitzDistribution::RoughnessToAlpha(rough);
MicrofacetDistribution *distrib =
    ARENA_ALLOC(arena, TrowbridgeReitzDistribution)(rough, rough);
```

9.2.3 MIX MATERIAL

It's useful to be able to combine two Materials with varying weights. The `MixMaterial` takes two other Materials and a `Spectrum`-valued texture and uses the `Spectrum` returned by the texture to blend between the two materials at the point being shaded. It is defined in the files `materials/mixmat.h` and `materials/mixmat.cpp`.

```
(MixMaterial Declarations) ≡
class MixMaterial : public Material {
public:
    (MixMaterial Public Methods 582)
private:
    (MixMaterial Private Data 582)
};
```

(MixMaterial Public Methods) ≡

```
MixMaterial(const std::shared_ptr<Material> &m1,
            const std::shared_ptr<Material> &m2,
            const std::shared_ptr<Texture<Spectrum>> &scale)
: m1(m1), m2(m2), scale(scale) {}
```

582

(MixMaterial Private Data) ≡

```
std::shared_ptr<Material> m1, m2;
std::shared_ptr<Texture<Spectrum>> scale;
```

582

(MixMaterial Method Definitions) ≡

```
void MixMaterial::ComputeScatteringFunctions(SurfaceInteraction *si,
                                             MemoryArena &arena, TransportMode mode,
                                             bool allowMultipleLobes) const {
    (Compute weights and original BxDFs for mix material 582)
    (Initialize si->bsdf with weighted mixture of BxDFs 583)
}
```

`MixMaterial::ComputeScatteringFunctions()` starts with its two constituent Materials initializing their respective BSDFs.

(Compute weights and original BxDFs for mix material) ≡

```
Spectrum s1 = scale->Evaluate(*si).Clamp();
Spectrum s2 = (Spectrum(1.f) - s1).Clamp();
SurfaceInteraction si2 = *si;
m1->ComputeScatteringFunctions(si, arena, mode, allowMultipleLobes);
m2->ComputeScatteringFunctions(&si2, arena, mode, allowMultipleLobes);
```

582

BSDF 572
 BxDF 513
 Material 577
`Material::ComputeScatteringFunctions()`
 577
 MemoryArena 1074
 MixMaterial 582
`MixMaterial::m1` 582
`MixMaterial::m2` 582
`MixMaterial::scale` 582
 ScaledBxDF 515
 Spectrum 315
`Spectrum::Clamp()` 317
 SurfaceInteraction 116
 Texture 614
`Texture::Evaluate()` 615
 TransportMode 960

It then scales BxDFs in the BSDF from the first material, `b1`, using the `ScaledBxDF` adapter class, and then scales the BxDFs from the second BSDF, adding all of these BxDF components to `si->bsdf`.

It may appear that there's a lurking memory leak in this code, in that the BxDF *s in `si->bdfs` are clobbered by newly allocated `ScaledBxDF`s. However, recall that those BxDFs, like the new ones here, were allocated through a `MemoryArena` and thus their memory will be freed when the `MemoryArena` frees its entire block of memory.

```
<Initialize si->bsdf with weighted mixture of BxDFs> ≡ 582
    int n1 = si->bsdf->NumComponents(), n2 = si2.bsdf->NumComponents();
    for (int i = 0; i < n1; ++i)
        si->bsdf->bxdfs[i] =
            ARENA_ALLOC(arena, ScaledBxDF)(si->bsdf->bxdfs[i], s1);
    for (int i = 0; i < n2; ++i)
        si->bsdf->Add(ARENA_ALLOC(arena, ScaledBxDF)(si2.bsdf->bxdfs[i], s2));
```

The implementation of `MixMaterial::ComputeScatteringFunctions()` needs direct access to the `bxdfs` member variables of the `BSDF` class. Because this is the only class that needs this access, we've just made `MixMaterial` a friend of `BSDF` rather than adding a number of accessor and setting methods.

```
(BSDF Private Data) +≡ 572
    friend class MixMaterial;
```

9.2.4 FOURIER MATERIAL

The `FourierMaterial` class supports measured or synthetic BSDF data that has been tabulated into the directional basis that was introduced in Section 8.6. It is defined in the files `materials/fourier.h` and `materials/fourier.cpp`.

```
<FourierMaterial Declarations> ≡
    class FourierMaterial : public Material {
    public:
        <FourierMaterial Public Methods>
    private:
        <FourierMaterial Private Data 583>
    };
```

The constructor is responsible for reading the BSDF from a file and initializing the `FourierBSDFTable`.

`ARENA_ALLOC()` 576
`BSDF` 572
`BSDF::bxdfs` 573
`BSDF::NumComponents()` 573
`Float` 1062
`FourierBSDF` 555
`FourierBSDFTable` 554
`FourierBSDFTable::Read()` 554
`FourierMaterial` 583
`Material` 577
`MixMaterial` 582
`ScaledBxDF` 515
`SurfaceInteraction::bsdf` 250
`Texture` 614

```
<FourierMaterial Method Definitions> ≡
    FourierMaterial::FourierMaterial(const std::string &filename,
        const std::shared_ptr<Texture<Float>> &bumpMap)
        : bumpMap(bumpMap) {
        FourierBSDFTable::Read(filename, &bsdfTable);
    }

<FourierMaterial Private Data> ≡ 583
    FourierBSDFTable bsdfTable;
    std::shared_ptr<Texture<Float>> bumpMap;
```

Once the data is in memory, the `ComputeScatteringFunctions()` method's task is straightforward. After the usual bump-mapping computation, it just has to allocate a `FourierBSDF` and provide it access to the data in the table.

```
(FourierMaterial Method Definitions) +≡
void FourierMaterial::ComputeScatteringFunctions(SurfaceInteraction *si,
    MemoryArena &arena, TransportMode mode,
    bool allowMultipleLobes) const {
(Perform bump mapping with bumpMap, if present 579)
    si->bsdf = ARENA_ALLOC(arena, BSDF)(*si);
    si->bsdf->Add(ARENA_ALLOC(arena, FourierBSDF)(bsdfTable, mode));
}
```

9.2.5 ADDITIONAL MATERIALS

Beyond these materials, there are eight more `Material` implementations available in `pbrt`, all in the `materials/` directory. We will not show all of their implementations here, since they are all just variations on the basic themes introduced in the material implementations above. All take `Textures` that define scattering parameters, these textures are evaluated in the materials' respective `ComputeScatteringFunctions()` methods, and appropriate `BxDFs` are created and returned in a `BSDF`. See the documentation on `pbrt`'s file format for a summary of the parameters that these materials take.

These materials include:

- `GlassMaterial`: Perfect or glossy specular reflection and transmission, weighted by Fresnel terms for accurate angular-dependent variation.
- `MetalMaterial`: Metal, based on the Fresnel equations for conductors and the Torrance–Sparrow model. Unlike plastic, metal includes no diffuse component. See the files in the directory `scenes/spds/metals/` for measured spectral data for the indices of refraction η and absorption coefficients k for a variety of metals.
- `MirrorMaterial`: A simple mirror, modeled with perfect specular reflection.
- `SubstrateMaterial`: A layered model that varies between glossy specular and diffuse reflection depending on the viewing angle (based on the `FresnelBlend` BRDF).
- `SubsurfaceMaterial` and `KdSubsurfaceMaterial`: Materials that return `BSSRDFs` that describe materials that exhibit subsurface scattering.
- `TranslucentMaterial`: A material that describes diffuse and glossy specular reflection and transmission through the surface.
- `UberMaterial`: A “kitchen sink” material representing the union of many of the preceding materials. This is a highly parameterized material that is particularly useful when converting scenes from other file formats into `pbrt`'s.

Figure 8.10 in the previous chapter shows the dragon model rendered with `GlassMaterial`, and Figure 9.4 shows it with the `MetalMaterial`. Figure 9.5 demonstrates the `KdSubsurfaceMaterial`.

9.3 BUMP MAPPING

All of the `Materials` defined in the previous section take an optional floating-point texture that defines a displacement at each point on the surface: each point p has a displaced point p' associated with it, defined by $p' = p + d(p)\mathbf{n}(p)$, where $d(p)$ is the offset returned by the displacement texture at p and $\mathbf{n}(p)$ is the surface normal at p (Figure 9.6).

ARENA_ALLOC() 576
 BSDF 572
 BxDF 513
 FourierBSDF 555
 FourierMaterial::bsdfTable 583
 GlassMaterial 584
 Material 577
 MemoryArena 1074
 MetalMaterial 584
 SurfaceInteraction 116
 SurfaceInteraction::bsdf 250
 Texture 614
 TransportMode 960



Figure 9.4: Dragon rendered with the `MetalMaterial1`, based on realistic measured gold scattering data. (Model courtesy of Christian Schüller.)



Figure 9.5: Head model rendered with the `KdSubsurfaceMaterial1`, which models subsurface scattering (in conjunction with the subsurface scattering light transport techniques from Section 15.5). (Model courtesy of Infinite Realities, Inc.)

We would like to use this texture to compute shading normals so that the surface appears as if it actually had been offset by the displacement function, without modifying its geometry. This process is called *bump mapping*. For relatively small displacement functions, the visual effect of bump mapping can be quite convincing. This idea and

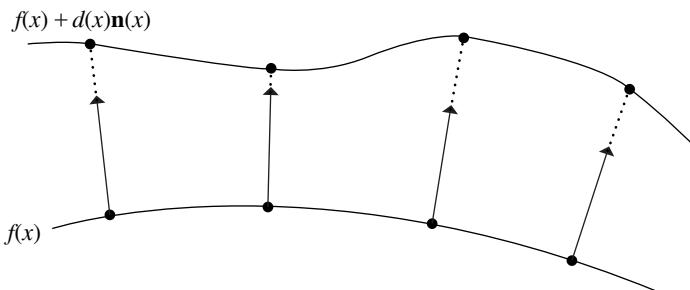


Figure 9.6: A displacement function associated with a material defines a new surface based on the old one, offset by the displacement amount along the normal at each point. pbrt doesn't compute a geometric representation of this displaced surface, but instead uses it to compute shading normals for bump mapping.

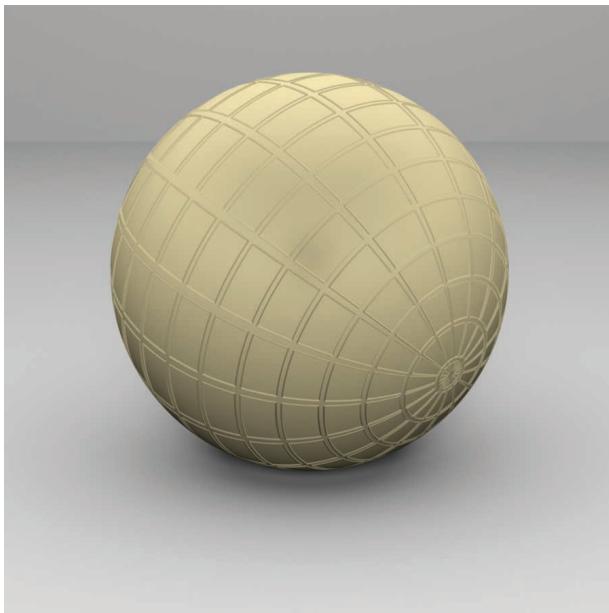


Figure 9.7: Using bump mapping to compute the shading normals for a sphere gives it the appearance of having much more geometric detail than is actually present.

the specific technique to compute these shading normals in a way that gives a plausible appearance of the actual displaced surface were developed by Blinn (1978).

Figure 9.7 shows the effect of applying bump mapping defined by an image map of a grid of lines to a sphere. A more complex example is shown in Figure 9.8, which shows a scene rendered with and without bump mapping. There, the bump map gives the appearance



(a)



(b)

Figure 9.8: The Sponza atrium model, rendered (a) without bump mapping and (b) with bump mapping. Bump mapping substantially increases the apparent geometric complexity of the model, without the increased rendering time and memory use that would result from a geometric representation with the equivalent amount of small-scale detail.

of a substantial amount of detail in the walls and floors that isn't actually present in the geometric model. Figure 9.9 shows one of the image maps used to define the bump function in Figure 9.8.

The `Material::Bump()` method is a utility routine for use by `Material` implementations. It is responsible for computing the effect of bump mapping at the point being shaded given a particular displacement `Texture`. So that future `Material` implementations aren't required to support bump mapping with this particular mechanism (or at all), we've placed this method outside of the hard-coded material evaluation pipeline and left it as a function that particular material implementations can call on their own.

Material 577

Material::Bump() 589

Texture 614



Figure 9.9: One of the image maps used as a bump map for the Sponza atrium rendering in Figure 9.8.

The implementation of `Material::Bump()` is based on finding an approximation to the partial derivatives $\partial p/\partial u$ and $\partial p/\partial v$ of the displaced surface and using them in place of the surface's actual partial derivatives to compute the shading normal. (Recall that the surface normal is given by the cross product of these vectors, $\mathbf{n} = \partial p/\partial u \times \partial p/\partial v$.) Assume that the original surface is defined by a parametric function $p(u, v)$, and the bump offset function is a scalar function $d(u, v)$. Then the displaced surface is given by

$$p'(u, v) = p(u, v) + d(u, v)\mathbf{n}(u, v),$$

where $\mathbf{n}(u, v)$ is the surface normal at (u, v) .

The partial derivatives of this function can be found using the chain rule. For example, the partial derivative in u is

$$\frac{\partial p'}{\partial u} = \frac{\partial p(u, v)}{\partial u} + \frac{\partial d(u, v)}{\partial u}\mathbf{n}(u, v) + d(u, v)\frac{\partial \mathbf{n}(u, v)}{\partial u}. \quad (9.1)$$

We already have computed the value of $\partial p(u, v)/\partial u$; it's $\partial p/\partial u$ and is available in the `SurfaceInteraction` structure, which also stores the surface normal $\mathbf{n}(u, v)$ and the partial derivative $\partial \mathbf{n}(u, v)/\partial u = \partial \mathbf{n}/\partial u$. The displacement function $d(u, v)$ can be evaluated as needed, which leaves $\partial d(u, v)/\partial u$ as the only remaining term.

There are two possible approaches to finding the values of $\partial d(u, v)/\partial u$ and $\partial d(u, v)/\partial v$. One option would be to augment the `Texture` interface with a method to compute partial derivatives of the underlying texture function. For example, for image map textures mapped to the surface directly using its (u, v) parameterization, these partial derivatives can be computed by subtracting adjacent texels in the u and v directions. However, this approach is difficult to extend to complex procedural textures like some of the ones defined in Chapter 10. Therefore, `pbrt` directly computes these values with forward differencing in the `Material::Bump()` method, without modifying the `Texture` interface.

`Material::Bump()` 589

`SurfaceInteraction` 116

`Texture` 614

Recall the definition of the partial derivative:

$$\frac{\partial d(u, v)}{\partial u} = \lim_{\Delta_u \rightarrow 0} \frac{d(u + \Delta_u, v) - d(u, v)}{\Delta_u}.$$

Forward differencing approximates the value using a finite value of Δ_u and evaluating $d(u, v)$ at two positions. Thus, the final expression for $\partial p'/\partial u$ is the following (for simplicity, we have dropped the explicit dependence on (u, v) for some of the terms):

$$\frac{\partial p'}{\partial u} \approx \frac{\partial p}{\partial u} + \frac{d(u + \Delta_u, v) - d(u, v)}{\Delta_u} \mathbf{n} + d(u, v) \frac{\partial \mathbf{n}}{\partial u}.$$

Interestingly enough, most bump-mapping implementations ignore the final term under the assumption that $d(u, v)$ is expected to be relatively small. (Since bump mapping is mostly useful for approximating small perturbations, this is a reasonable assumption.) The fact that many renderers do not compute the values $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$ may also have something to do with this simplification. An implication of ignoring the last term is that the magnitude of the displacement function then does not affect the bump-mapped partial derivatives; adding a constant value to it globally doesn't affect the final result, since only differences of the bump function affect it. pbrt computes all three terms since it has $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$ readily available, although in practice this final term rarely makes a visually noticeable difference.

One important detail in the definition of `Bump()` is that the `d` parameter is declared to be of type `const shared_ptr<Texture<Float>>` &, rather than, for example, `shared_ptr<Texture<Float>>`. This difference is very important for performance, but the reason is subtle. If a C++ reference was not used here, then the `shared_ptr` implementation would need to increment the reference count for the temporary value passed to the method, and the reference count would need to be decremented when the method returned. This is an efficient operation with serial code, but with multiple threads of execution, it leads to a situation where multiple processing cores end up modifying the same memory location whenever different rendering tasks run this method with the same displacement texture. This state of affairs in turn leads to the expensive “read for ownership” operation described in Section A.6.1.²

{Material Method Definitions} ≡

```
void Material::Bump(const std::shared_ptr<Texture<Float>> &d,
                     SurfaceInteraction *si) {
    {Compute offset positions and evaluate displacement texture 590}
    {Compute bump-mapped differential geometry 590}
}
```

Float 1062

Material 577

SurfaceInteraction 116

Texture 614

2 As with many of the asides like this one, the underlying lesson was learned painfully by the authors: when we multi-threaded pbrt while working on the second edition of the book, we got this wrong. The issue was only discovered some months later in development when profiling the system revealed that calling `Bump()` was taking a surprising amount of time.

$\langle \text{Compute offset positions and evaluate displacement texture} \rangle \equiv$ <pre>SurfaceInteraction siEval = *si; (Shift siEval du in the u direction 590) Float uDisplace = d->Evaluate(siEval); (Shift siEval dv in the v direction) Float vDisplace = d->Evaluate(siEval); Float displace = d->Evaluate(*si);</pre>	589
---	-----

One remaining issue is how to choose the offsets Δ_u and Δ_v for the finite differencing computations. They should be small enough that fine changes in $d(u, v)$ are captured but large enough so that available floating-point precision is sufficient to give a good result. Here, we will choose Δ_u and Δ_v values that lead to an offset that is about half the image space pixel sample spacing and use them to update the appropriate member variables in the SurfaceInteraction to reflect a shift to the offset position. (See Section 10.1.1 for an explanation of how the image space distances are computed.)

Another detail to note in the following code: we recompute the surface normal \mathbf{n} as the cross product of $\partial p / \partial u$ and $\partial p / \partial v$ rather than using $si->\text{shading}.n$ directly. The reason for this is that the orientation of \mathbf{n} may have been flipped (recall the fragment *(Adjust normal based on orientation and handedness)* in Section 2.10.1). However, we need the original normal here. Later, when the results of the computation are passed to `SurfaceInteraction::SetShadingGeometry()`, the normal we compute will itself be flipped if necessary.

$\langle \text{Shift siEval du in the u direction} \rangle \equiv$ <pre>Float du = .5f * (std::abs(si->dudx) + std::abs(si->dudy)); if (du == 0) du = .01f; siEval.p = si->p + du * si->shading.dpdu; siEval.uv = si->uv + Vector2f(du, 0.f); siEval.n = Normalize((Normal3f)Cross(si->shading.dpdu, si->shading.dpdv) + du * si->dndu);</pre>	590
---	-----

The *(Shift siEval dv in the v direction)* fragment is nearly the same as the fragment that shifts du , so it isn't included here.

Given the new positions and the displacement texture's values at them, the partial derivatives can be computed directly using Equation (9.1):

$\langle \text{Compute bump-mapped differential geometry} \rangle \equiv$ <pre>Vector3f dpdu = si->shading.dpdu + (uDisplace - displace) / du * Vector3f(si->shading.n) + displace * Vector3f(si->shading.dndu); Vector3f dpdv = si->shading.dpdv + (vDisplace - displace) / dv * Vector3f(si->shading.n) + displace * Vector3f(si->shading.dndv); si->SetShadingGeometry(dpdu, dpdv, si->shading.dndu, si->shading.dndv, false);</pre>	589
--	-----

```
Float 1062
Interaction::n 116
Interaction::p 115
Normal3::Normalize() 71
Normal3f 71
SurfaceInteraction 116
SurfaceInteraction::dndu 116
SurfaceInteraction::dudx 600
SurfaceInteraction::dudy 600
SurfaceInteraction::SetShadingGeometry()
    119
SurfaceInteraction::shading
    118
SurfaceInteraction::
    shading::dndu
    118
SurfaceInteraction::
    shading::dndv
    118
SurfaceInteraction::
    shading::dpdu
    118
SurfaceInteraction::
    shading::dpdv
    118
SurfaceInteraction::
    shading::n
    118
SurfaceInteraction::uv 116
Texture::Evaluate() 615
Vector2f 60
Vector3f 60
```

FURTHER READING

Burley's article (2012) on a material model developed at Disney for feature films is an excellent read. It includes extensive discussion of features of real-world reflection functions that can be observed in Matusik et al.'s (2003b) measurements of one hundred BRDFs and analyzes the ways that existing BRDF models do and do not fit these features well. These insights are then used to develop an "artist-friendly" material model that can express a wide range of surface appearances. The model describes reflection with a single color and ten scalar parameters, all of which are in the range [0, 1] and have fairly predictable effects on the appearance of the resulting material.

Blinn (1978) invented the bump-mapping technique. Kajiya (1985) generalized the idea of bump mapping the normal to *frame mapping*, which also perturbs the surface's primary tangent vector and is useful for controlling the appearance of anisotropic reflection models. Mikkelsen's thesis (2008) carefully investigates a number of the assumptions underlying bump mapping, proposes generalizations, and addresses a number of subtleties with respect to its application to real-time rendering.

Snyder and Barr (1987) noted the light leak problem from per-vertex shading normals and proposed a number of work-arounds. The method we have used in this chapter is from Section 5.3 of Veach's thesis (1997); it is a more robust solution than those of Snyder and Barr.

Shading normals introduce a number of subtle problems for physically based light transport algorithms that we have not addressed in this chapter. For example, they can easily lead to surfaces that reflect more energy than was incident upon them, which can wreak havoc with light transport algorithms that are designed under the assumption of energy conservation. Veach (1996) investigated this issue in depth and developed a number of solutions. Section 16.1 of this book will return to this issue.

One visual shortcoming of bump mapping is that it doesn't naturally account for self-shadowing, where bumps cast shadows on the surface and prevent light from reaching nearby points. These shadows can have a significant impact on the appearance of rough surfaces. Max (1988) developed the *horizon mapping* technique, which performs a preprocess on bump maps stored in image maps to compute a term to account for this effect. This approach isn't directly applicable to procedural textures, however. Dana et al. (1999) measured spatially varying reflection properties from real-world surfaces, including these self-shadowing effects; they convincingly demonstrate this effect's importance for accurate image synthesis.

Another difficult issue related to bump mapping is that antialiasing bump maps that have higher frequency detail than can be represented in the image is quite difficult. In particular, it is not enough to remove high-frequency detail from the bump map function, but in general the BSDF needs to be modified to account for this detail. Fournier (1992) applied *normal distribution functions* to this problem, where the surface normal was generalized to represent a distribution of normal directions. Becker and Max (1993) developed algorithms for blending between bump maps and BRDFs that represented higher-frequency

details. Schilling (1997, 2001) investigated this issue particularly for application to graphics hardware. More recently, effective approaches to filtering bump maps were developed by Han et al. (2007) and Olano and Baker (2010). Recent work by Dupuy et al. (2013) and Hery et al. (2014) addressed this issue by developing techniques that convert displacements into anisotropic distributions of Beckmann microfacets.

An alternative to bump mapping is displacement mapping, where the bump function is used to actually modify the surface geometry, rather than just perturbing the normal (Cook 1984; Cook, Carpenter, and Catmull 1987). Advantages of displacement mapping include geometric detail on object silhouettes and the possibility of accounting for self-shadowing. Patterson and collaborators described an innovative algorithm for displacement mapping with ray tracing where the geometry is unperturbed but the ray's direction is modified such that the intersections that are found are the same as would be found with the displaced geometry (Patterson, Hoggar, and Logie 1991; Logie and Patterson 1994). Heidrich and Seidel (1998) developed a technique for computing direct intersections with procedurally defined displacement functions.

As computers have become faster, another viable approach for displacement mapping has been to use an implicit function to define the displaced surface and to then take steps along rays until they find a zero crossing with the implicit function. At this point, an intersection has been found. This approach was first introduced by Hart (1996); see Donnelly (2005) for information about using this approach for displacement mapping on the GPU. This approach was recently popularized by Quilez on the *shadertoy* Web site (Quilez 2015).

With the advent of increased memory on computers and caching algorithms, the option of finely tessellating geometry and displacing its vertices for ray tracing has become feasible. Pharr and Hanrahan (1996) described an approach to this problem based on geometry caching, and Wang et al. (2000) described an adaptive tessellation algorithm that reduces memory requirements. Smits, Shirley, and Stark (2000) lazily tessellate individual triangles, saving a substantial amount of memory.

Measuring fine-scale surface geometry of real surfaces to acquire bump or displacement maps can be challenging. Johnson et al. (2011) developed a novel hand-held system that can measure detail down to a few microns, which more than suffices for these uses.

EXERCISES

- ➊ 9.1 If the same `Texture` is bound to more than one component of a `Material` (e.g., to both `PlasticMaterial::Kd` and `PlasticMaterial::Ks`), the texture will be evaluated twice. This unnecessarily duplicated work may lead to a noticeable increase in rendering time if the `Texture` is itself computationally expensive. Modify the materials in `pbrt` to eliminate this problem. Measure the change in the system's performance, both for standard scenes as well as for contrived scenes that exhibit this redundancy.
- ➋ 9.2 Implement the artist-friendly “Disney BRDF” described by Burley (2012). You will need both a new `Material` implementation as well as a few new `BxDFs`. Ren-

`BxDF` 513

`Material` 577

`PlasticMaterial::Kd` 580

`PlasticMaterial::Ks` 580

`Texture` 614

der a variety of scenes using your implementation. How easy do you find it to match the visual appearance of existing `pbrt` scenes when replacing `Materials` in them with this one? How quickly can you dial in the parameters of this material to achieve a given desired appearance?

- ③ 9.3 One form of aliasing that `pbrt` doesn't try to eliminate is specular highlight aliasing. Glossy specular surfaces with high specular exponents, particularly if they have high curvature, are susceptible to aliasing as small changes in incident direction or surface position (and thus surface normal) may cause the highlight's contribution to change substantially. Read Amanatides's paper on this topic (Amanatides 1992) and extend `pbrt` to reduce specular aliasing, either using his technique or by developing your own. Most of the quantities needed to do the appropriate computations are already available— $\partial\mathbf{n}/\partial x$ and $\partial\mathbf{n}/\partial y$ in the `SurfaceInteraction`, and so on—although it will probably be necessary to extend the `BxDF` interface to provide more information about the roughness of any `MicrofacetDistributions` they have.
- ② 9.4 Another approach to addressing specular highlight aliasing is to supersample the `BSDF`, evaluating it multiple times around the point being shaded. After reading the discussion of supersampling texture functions in Section 10.1, modify the `BSDF::f()` method to shift to a set of positions around the intersection point but within the pixel sampling rate around the intersection point and evaluate the `BSDF` at each one of them when the `BSDF` evaluation routines are called. (Be sure to account for the change in normal using its partial derivatives.) How well does this approach combat specular highlight aliasing?
- ③ 9.5 Read some of the papers on filtering bump maps referenced in the “Further Reading” section of this chapter, choose one of the techniques described there, and implement it in `pbrt`. Show both the visual artifacts from bump map aliasing without the technique you implement as well as examples of how well your implementation addresses them.
- ③ 9.6 Neyret (1996, 1998), Heitz and Neyret (2012), and Heitz et al. (2015) developed algorithms that take descriptions of complex shapes and their reflective properties and turn them into generalized reflection models at different resolutions, each with limited frequency content. The advantage of this representation is that it makes it easy to select an appropriate level of detail for an object based on its size on the screen, thus reducing aliasing. Read these papers and implement the algorithms described in them in `pbrt`. Show how they can be used to reduce geometric aliasing from detailed geometry, and extend them to address bump map aliasing.
- ③ 9.7 Use the triangular face refinement infrastructure from the Loop subdivision surface implementation in Section 3.8 to implement displacement mapping in `pbrt`. The usual approach to displacement mapping is to finely tessellate the geometric shape and then to evaluate the displacement function at its vertices, moving each vertex the given distance along its normal.

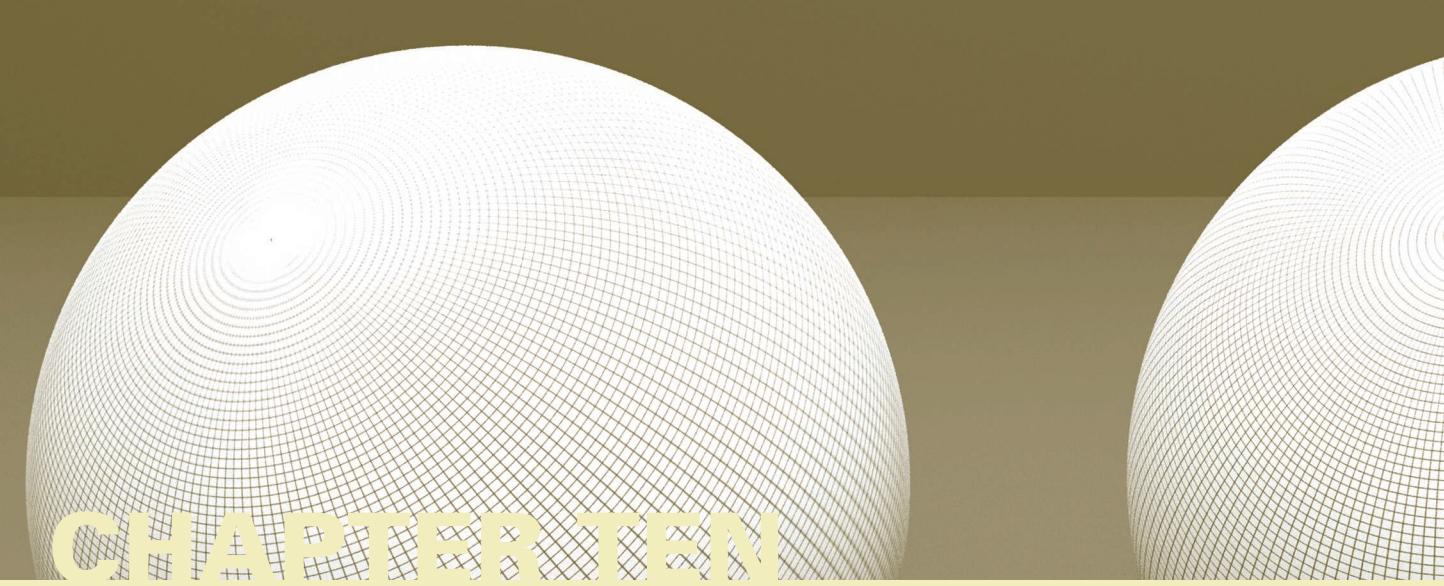
`BSDF::f()` 575

`BxDF` 513

`SurfaceInteraction` 116

Refine each face of the mesh until, when projected onto the image, it is roughly the size of the separation between pixels. To do this, you will need to be able to estimate the image pixel-based length of an edge in the scene when it is projected onto the screen. Use the texturing infrastructure in Chapter 10 to evaluate displacement functions. See Patney et al. (2009) and Fisher et al. (2009) for discussion of issues related to avoiding cracks in the mesh due to adaptive tessellation.

This page intentionally left blank



CHAPTER TEN

10 TEXTURE

We will now describe a set of interfaces and classes that allow us to incorporate texture into our material models. Recall that the materials in Chapter 9 are all based on various parameters that describe their characteristics (diffuse reflectance, glossiness, etc.). Because real-world material properties typically vary over surfaces, it is necessary to be able to represent this spatial variation. In pbrt, the texture abstractions serve this purpose. They are defined in a way that separates the pattern generation methods from the reflection model implementations, making it easy to combine them in arbitrary ways, thereby making it easier to create a wide variety of appearances.

In pbrt, a *texture* is a fairly general concept: it is a function that maps points in some domain (e.g., a surface’s (u, v) parametric space or (x, y, z) object space) to values in some other domain (e.g., spectra or the real numbers). A variety of implementations of texture classes are available in the system. For example, pbrt has textures that represent zero-dimensional functions that return a constant in order to accommodate surfaces that have the same parameter value everywhere. Image map textures are two-dimensional functions of (s, t) parameter values that use a 2D array of pixel values to compute texture values at particular points (they are described in Section 10.4). There are even texture functions that compute values based on the values computed by other texture functions.

Textures may be a source of high-frequency variation in the final image. Figure 10.1 shows an image with severe aliasing due to a texture. Although the visual impact of this aliasing can be reduced with the nonuniform sampling techniques from Chapter 7, a better solution to this problem is to implement texture functions that adjust their frequency content based on the rate at which they are being sampled. For many texture functions, computing a reasonable approximation to the frequency content and antialiasing in this manner aren’t too difficult and are substantially more efficient than reducing aliasing by increasing the image sampling rate.

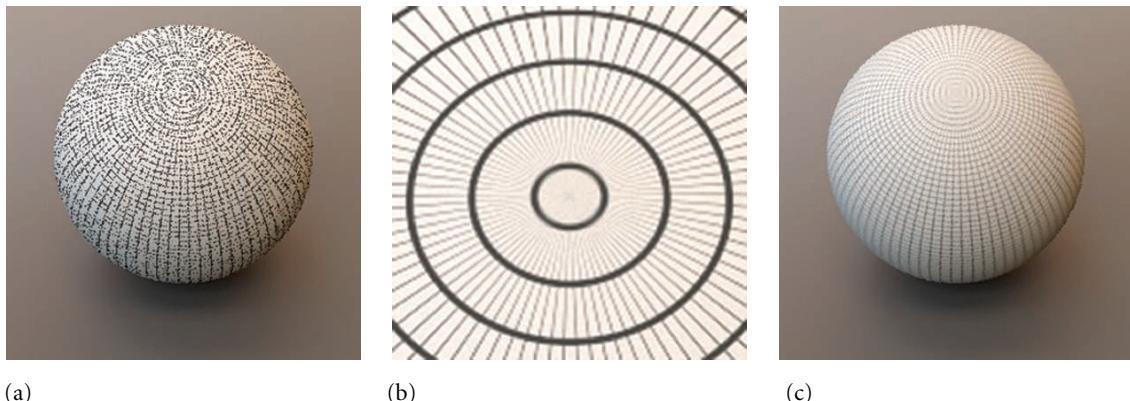


Figure 10.1: Texture Aliasing. (a) An image of a grid texture on a sphere with one sample per pixel has severe aliasing artifacts. (b) A zoomed-in area from near the top of the sphere gives a sense of how much high-frequency detail is present between adjacent pixel sample positions. (c) The texture function has taken into account the image sampling rate to prefilter its function and remove high-frequency detail, resulting in an antialiased image, even with a single sample per pixel.

The first section of this chapter will discuss the problem of texture aliasing and general approaches to solving it. We will then describe the basic texture interface and illustrate its use with a few simple texture functions. Throughout the remainder of the chapter, we will present a variety of more complex texture implementations, demonstrating the use of a number of different texture antialiasing techniques along the way.

10.1 SAMPLING AND ANTIALIASING

The sampling task from Chapter 7 was a frustrating one since the aliasing problem was known to be unsolvable from the start. The infinite frequency content of geometric edges and hard shadows *guarantees* aliasing in the final images, no matter how high the image sampling rate. (Our only consolation is that the visual impact of this remaining aliasing can be reduced to unobjectionable levels with a sufficient number of well-placed samples.)

Fortunately, for textures things are not this difficult from the start: either there is often a convenient analytic form of the texture function available, which makes it possible to remove excessively high frequencies before sampling it, or it is possible to be careful when evaluating the function so as not to introduce high frequencies in the first place. When this problem is carefully addressed in texture implementations, as is done through the rest of this chapter, there is usually no need for more than one sample per pixel in order to render an image without texture aliasing.

Two problems must be addressed in order to remove aliasing from texture functions:

1. The sampling rate in texture space must be computed. The screen space sampling rate is known from the image resolution and pixel sampling rate, but here we need to determine the resulting sampling rate on a surface in the scene in order to find the rate at which the texture function is being sampled.

- Given the texture sampling rate, sampling theory must be applied to guide the computation of a texture value that doesn't have higher frequency variation than can be represented by the sampling rate (e.g., by removing excess frequencies beyond the Nyquist limit from the texture function).

These two issues will be addressed in turn throughout the rest of this section.

10.1.1 FINDING THE TEXTURE SAMPLING RATE

Consider an arbitrary texture function that is a function of position, $T(p)$, defined on a surface in the scene. If we ignore the complications introduced by visibility issues—the possibility that another object may occlude the surface at nearby image samples or that the surface may have a limited extent on the image plane—this texture function can also be expressed as a function over points (x, y) on the image plane, $T(f(x, y))$, where $f(x, y)$ is the function that maps image points to points on the surface. Thus, $T(f(x, y))$ gives the value of the texture function as seen at image position (x, y) .

As a simple example of this idea, consider a 2D texture function $T(s, t)$ applied to a quadrilateral that is perpendicular to the z axis and has corners at the world space points $(0, 0, 0)$, $(1, 0, 0)$, $(1, 1, 0)$, and $(0, 1, 0)$. If an orthographic camera is placed looking down the z axis such that the quadrilateral precisely fills the image plane and if points p on the quadrilateral are mapped to 2D (s, t) texture coordinates by

$$s = p_x \quad t = p_y,$$

then the relationship between (s, t) and screen (x, y) pixels is straightforward:

$$s = \frac{x}{x_r} \quad t = \frac{y}{y_r},$$

where the overall image resolution is (x_r, y_r) (Figure 10.2). Thus, given a sample spacing of one pixel in the image plane, the sample spacing in (s, t) texture parameter space is $(1/x_r, 1/y_r)$, and the texture function must remove any detail at a higher frequency than can be represented at that sampling rate.

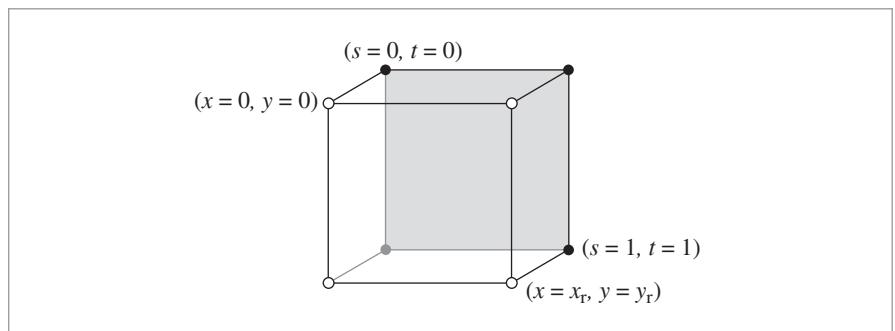


Figure 10.2: If a quadrilateral is viewed with an orthographic perspective such that the quadrilateral precisely fills the image plane, it's easy to compute the relationship between the sampling rate in (x, y) pixel coordinates and the texture sampling rate.

This relationship between pixel coordinates and texture coordinates, and thus the relationship between their sampling rates, is the key bit of information that determines the maximum frequency content allowable in the texture function. As a slightly more complex example, given a triangle with (s, t) texture coordinates at the vertices and viewed with a perspective projection, it's possible to analytically find the differences in s and t across the sample points on the image plane. This is the basis of basic texture map antialiasing in graphics processors.

For more complex scene geometry, camera projections, and mappings to texture coordinates, it is much more difficult to precisely determine the relationship between image positions and texture parameter values. Fortunately, for texture antialiasing, we don't need to be able to evaluate $f(x, y)$ for arbitrary (x, y) but just need to find the relationship between changes in pixel sample position and the resulting change in texture sample position at a particular point on the image. This relationship is given by the partial derivatives of this function, $\partial f / \partial x$ and $\partial f / \partial y$. For example, these can be used to find a first-order approximation to the value of f ,

$$f(x', y') \approx f(x, y) + (x' - x) \frac{\partial f}{\partial x} + (y' - y) \frac{\partial f}{\partial y}.$$

If these partial derivatives are changing slowly with respect to the distances $x' - x$ and $y' - y$, this is a reasonable approximation. More importantly, the values of these partial derivatives give an approximation to the change in texture sample position for a shift of one pixel in the x and y directions, respectively, and thus directly yield the texture sampling rate. For example, in the previous quadrilateral example, $\partial s / \partial x = 1/x_r$, $\partial s / \partial y = 0$, $\partial t / \partial x = 0$, and $\partial t / \partial y = 1/y_r$.

The key to finding the values of these partial derivatives in the general case lies in the `RayDifferential` structure, which was defined in Section 2.5.1. This structure is initialized for each camera ray by the `Camera::GenerateRayDifferential()` method; it contains not only the ray actually being traced through the scene but also two additional rays, one offset horizontally one pixel sample from the camera ray and the other offset vertically by one pixel sample. All of the geometric ray intersection routines use only the main camera ray for their computations; the auxiliary rays are ignored (this is easy to do because `RayDifferential` is a subclass of `Ray`).

Here we will use the offset rays to estimate the partial derivatives of the mapping $p(x, y)$ from image position to world space position and the partial derivatives of the mappings $u(x, y)$ and $v(x, y)$ from (x, y) to (u, v) parametric coordinates, giving the partial derivatives of world space positions $\partial p / \partial x$ and $\partial p / \partial y$ and the partial derivatives of (u, v) parametric coordinates $\partial u / \partial x$, $\partial v / \partial x$, $\partial u / \partial y$, and $\partial v / \partial y$. In Section 10.2, we will see how these can be used to compute the screen space derivatives of arbitrary quantities based on p or (u, v) and consequently the sampling rates of these quantities. The values of these partial derivatives at the intersection point are stored in the `SurfaceInteraction` structure. They are declared as `mutable`, since they are set in a method that takes a `const` instance of that object.

(SurfaceInteraction Public Data) +≡

```
mutable Vector3f dpdx, dpdy;
mutable Float dudx = 0, dvdx = 0, dudy = 0, dvdy = 0;
```

116

Camera:::
 GenerateRayDifferential()
 357

Float 1062

Ray 73

RayDifferential 75

SurfaceInteraction 116

Vector3f 60

The `SurfaceInteraction::ComputeDifferentials()` method computes these values. It is called by `SurfaceInteraction::ComputeScatteringFunctions()` before the Material's `ComputeScatteringFunctions()` method is called so that these values will be available for any texture evaluation routines that are called by the material. Because ray differentials aren't available for all rays traced by the system (e.g., rays starting from light sources traced for photon mapping or bidirectional path tracing), the `hasDifferentials` field of the `RayDifferential` must be checked before these computations are performed. If the differentials are not present, then the derivatives are all set to zero (which will eventually lead to unfiltered point sampling of textures).

```
<SurfaceInteraction Method Definitions> +≡
void SurfaceInteraction::ComputeDifferentials(
    const RayDifferential &ray) const {
    if (ray.hasDifferentials) {
        <Estimate screen space change in p and (u, v) 601>
    } else {
        dudx = dvdx = 0;
        dudy = dvdy = 0;
        dpdx = dpdy = Vector3f(0, 0, 0);
    }
}
```

The key to computing these estimates is the assumption that the surface is locally flat with respect to the sampling rate at the point being shaded. This is a reasonable approximation in practice, and it is hard to do much better. Because ray tracing is a point-sampling technique, we have no additional information about the scene in between the rays we have traced. For highly curved surfaces or at silhouette edges, this approximation can break down, though this is rarely a source of noticeable error in practice.

For this approximation, we need the plane through the point p intersected by the main ray that is tangent to the surface. This plane is given by the implicit equation

$$ax + by + cz + d = 0,$$

where $a = \mathbf{n}_x$, $b = \mathbf{n}_y$, $c = \mathbf{n}_z$, and $d = -(\mathbf{n} \cdot \mathbf{p})$. We can then compute the intersection points p_x and p_y between the auxiliary rays r_x and r_y and this plane (Figure 10.3). These new points give an approximation to the partial derivatives of position on the surface $\partial p / \partial x$ and $\partial p / \partial y$, based on forward differences:

$$\frac{\partial p}{\partial x} \approx p_x - p, \quad \frac{\partial p}{\partial y} \approx p_y - p.$$

Because the differential rays are offset one pixel sample in each direction, there's no need to divide these differences by a Δ value, since $\Delta = 1$.

```
Interaction::p 115
Material 577
RayDifferential 75
RayDifferential::  
    hasDifferentials  
    75
SurfaceInteraction 116
SurfaceInteraction::  
    ComputeDifferentials()  
    601
SurfaceInteraction::  
    ComputeScatteringFunctions()  
    578
SurfaceInteraction::dpdx 600
SurfaceInteraction::dpdy 600
SurfaceInteraction::dudx 600
SurfaceInteraction::dudy 600
SurfaceInteraction::dvdx 600
SurfaceInteraction::dvdy 600
Vector3f 60

```

<Estimate screen space change in p and (u, v)> ≡

<Compute auxiliary intersection points with plane 602>

`dpdx = px - p;`
`dpdy = py - p;`

<Compute (u, v) offsets at auxiliary points 603>

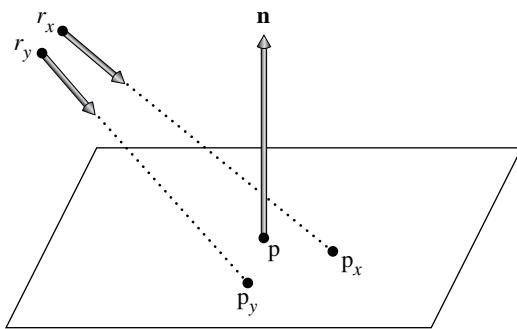


Figure 10.3: By approximating the local surface geometry at the intersection point with the tangent plane through p , approximations to the points at which the auxiliary rays r_x and r_y would intersect the surface can be found by finding their intersection points with the tangent plane p_x and p_y .

The ray–plane intersection algorithm described in Section 3.1.2 gives the t value where a ray described by origin \mathbf{o} and direction \mathbf{d} intersects a plane described by $ax + by + cz + d = 0$:

$$t = \frac{-((a, b, c) \cdot \mathbf{o}) - d}{(a, b, c) \cdot \mathbf{d}}.$$

To compute this value for the two auxiliary rays, the plane’s d coefficient is computed first. It isn’t necessary to compute the a , b , and c coefficients, since they’re available in \mathbf{n} . We can then apply the formula directly.

(Compute auxiliary intersection points with plane) \equiv

601

```
Float d = Dot(n, Vector3f(p.x, p.y, p.z));
Float tx = -(Dot(n, Vector3f(ray.rxOrigin)) - d) /
    Dot(n, ray.rxDirection);
Point3f px = ray.rxOrigin + tx * ray.rxDirection;
Float ty = -(Dot(n, Vector3f(ray.ryOrigin)) - d) /
    Dot(n, ray.ryDirection);
Point3f py = ray.ryOrigin + ty * ray.ryDirection;
```

Using the positions p_x and p_y , an approximation to their respective (u, v) coordinates can be found by taking advantage of the fact that the surface’s partial derivatives $\partial\mathbf{p}/\partial u$ and $\partial\mathbf{p}/\partial v$ form a (not necessarily orthogonal) coordinate system on the plane and that the coordinates of the auxiliary intersection points in terms of this coordinate system are their coordinates with respect to the (u, v) parameterization (Figure 10.4). Given a position \mathbf{p}' on the plane, we can compute its position with respect to the coordinate system by

$$\mathbf{p}' = \mathbf{p} + \Delta_u \frac{\partial \mathbf{p}}{\partial u} + \Delta_v \frac{\partial \mathbf{p}}{\partial v},$$

Dot() 63
Float 1062
Interaction::n 116
Point3f 68
RayDifferential::rxDirection 75
RayDifferential::ryOrigin 75
RayDifferential::ryDirection 75
Vector3f 60

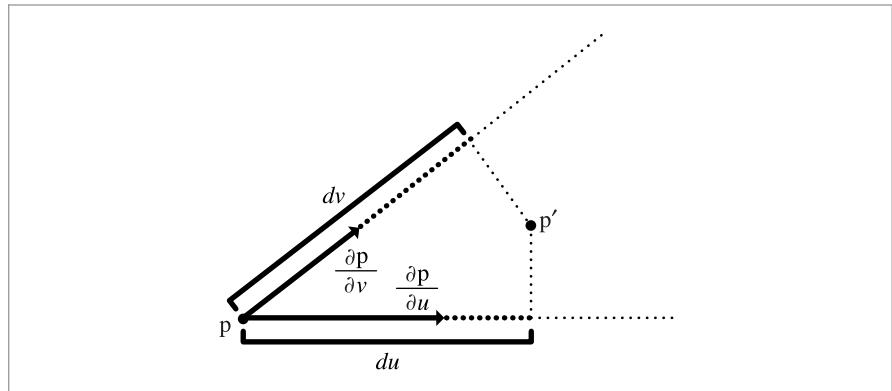


Figure 10.4: An estimate of the difference in (u, v) parametric coordinates from p to p' can be found by finding the coordinates of p' with respect to the coordinate system defined by p , $\partial p / \partial u$, and $\partial p / \partial v$.

or, equivalently,

$$\begin{pmatrix} p'_x - p_x \\ p'_y - p_y \\ p'_z - p_z \end{pmatrix} = \begin{pmatrix} \partial p_x / \partial u & \partial p_x / \partial v \\ \partial p_y / \partial u & \partial p_y / \partial v \\ \partial p_z / \partial u & \partial p_z / \partial v \end{pmatrix} \begin{pmatrix} \Delta_u \\ \Delta_v \end{pmatrix}.$$

A solution to this linear system of equations for one of the auxiliary points $p' = p_x$ or $p' = p_y$ gives the corresponding screen space partial derivatives $(\partial u / \partial x, \partial v / \partial x)$ or $(\partial u / \partial y, \partial v / \partial y)$, respectively.

This linear system has three equations with two unknowns—that is, it's overconstrained. We need to be careful since one of the equations may be degenerate—for example, if $\partial p / \partial u$ and $\partial p / \partial v$ are in the xy plane such that their z components are both zero, then the third equation will be degenerate. Therefore, we'd like to solve the system of equations using two equations that don't give a degenerate system. An easy way to do this is to take the cross product of $\partial p / \partial u$ and $\partial p / \partial v$, see which coordinate of the result has the largest magnitude, and use the other two. Their cross product is already available in n , so using this approach is straightforward. Even after all this, it may happen that the linear system has no solution (usually due to the partial derivatives not forming a coordinate system on the plane). In that case, all that can be done is to return arbitrary values.

(Compute (u, v) offsets at auxiliary points) ≡

601

```
SolveLinearSystem2x2() 1080
SurfaceInteraction::dudx 600
SurfaceInteraction::dudy 600
SurfaceInteraction::dvdx 600
SurfaceInteraction::dvyd 600
```

(Choose two dimensions to use for ray offset computation) ≡ 603

```
int dim[2];
if (std::abs(n.x) > std::abs(n.y) && std::abs(n.x) > std::abs(n.z)) {
    dim[0] = 1; dim[1] = 2;
} else if (std::abs(n.y) > std::abs(n.z)) {
    dim[0] = 0; dim[1] = 2;
} else {
    dim[0] = 0; dim[1] = 1;
}
```

(Initialize A, Bx, and By matrices for offset computation) ≡ 603

```
Float A[2][2] = { { dpdu[dim[0]], dpdv[dim[0]] },
                  { dpdu[dim[1]], dpdv[dim[1]] } };
Float Bx[2] = { px[dim[0]] - p[dim[0]], px[dim[1]] - p[dim[1]] };
Float By[2] = { py[dim[0]] - p[dim[0]], py[dim[1]] - p[dim[1]] };
```

10.1.2 FILTERING TEXTURE FUNCTIONS

It is necessary to remove frequencies in texture functions that are past the Nyquist limit for the texture sampling rate. The goal is to compute, with as few approximations as possible, the result of the *ideal texture resampling* process, which says that in order to evaluate $T(f(x, y))$ without aliasing, we must first band-limit it, removing frequencies beyond the Nyquist limit by convolving it with the sinc filter:

$$T'_b(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \text{sinc}(x') \text{sinc}(y') T'(f(x + x', y + y')) dx' dy'.$$

The band-limited function in turn should then be convolved with the pixel filter $g(x, y)$ centered at the (x, y) point on the screen at which we want to evaluate the texture function:

$$T'_f(x, y) = \int_{-yWidth/2}^{yWidth/2} \int_{-xWidth/2}^{xWidth/2} g(x', y') T'_b(x + x', y + y') dx' dy'.$$

This gives the theoretically perfect value for the texture as projected onto the screen.¹

In practice, there are many simplifications that can be made to this process, with little reduction in visual quality. For example, a box filter may be used for the band-limiting step, and the second step is usually ignored completely, effectively acting as if the pixel filter were a box filter, which makes it possible to do the antialiasing work completely in texture space and simplifies the implementation significantly. The EWA filtering algorithm in Section 10.4.5 is a notable exception in that it assumes a Gaussian pixel filter.

¹ One simplification that is present in this ideal filtering process is the implicit assumption that the texture function makes a linear contribution to frequency content in the image, so that filtering out its high frequencies removes high frequencies from the image. This is true for many uses of textures—for example, if an image map is used to modulate the diffuse term of a MatteMaterial. However, if a texture is used to determine the roughness of a glossy specular object, for example, this linearity assumption is incorrect, since a linear change in the roughness value has a non-linear effect on the reflected radiance from the microfacet BRDF. We will ignore this issue here, since it isn't easily solved in general. The “Further Reading” section has more discussion of this issue.

Float 1062

Interaction::n 116

Interaction::p 115

MatteMaterial 578

SurfaceInteraction::dpdu 116

SurfaceInteraction::dpdv 116

The box filter is easy to use, since it can be applied analytically by computing the average of the texture function over the appropriate region. Intuitively, this is a reasonable approach to the texture filtering problem, and it can be computed directly for many texture functions. Indeed, through the rest of this chapter, we will often use a box filter to average texture function values between samples and informally use the term *filter region* to describe the area being averaged over. This is the most common approach when filtering texture functions.

Even the box filter, with all of its shortcomings, gives acceptable results for texture filtering in many cases. One factor that helps is the fact that a number of samples are usually taken in each pixel. Thus, even if the filtered texture values used in each one are sub-optimal, once they are filtered by the pixel reconstruction filter, the end result generally doesn't suffer too much.

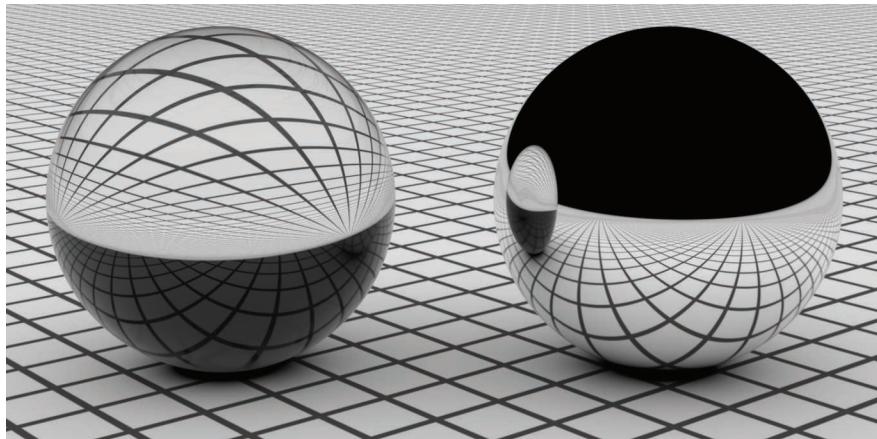
An alternative to using the box filter to filter texture functions is to use the observation that the effect of the ideal sinc filter is to let frequency components below the Nyquist limit pass through unchanged but to remove frequencies past it. Therefore, if we know the frequency content of the texture function (e.g., if it is a sum of terms, each one with known frequency content), then if we replace the high-frequency terms with their average values, we are effectively doing the work of the sinc prefilter. This approach is known as *clamping* and is the basis for antialiasing in the textures based on the noise function in Section 10.6.

Finally, for texture functions where none of these techniques is easily applied, a final option is *supersampling*—the function is evaluated and filtered at multiple locations near the main evaluation point, thus increasing the sampling rate in texture space. If a box filter is used to filter these sample values, this is equivalent to averaging the value of the function. This approach can be expensive if the texture function is complex to evaluate, and as with image sampling a very large number of samples may be needed to remove aliasing. Although this is a brute-force solution, it is still more efficient than increasing the image sampling rate, since it doesn't incur the cost of tracing more rays through the scene.

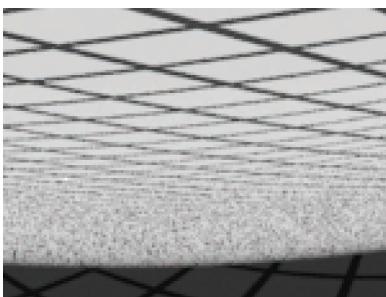
* 10.1.3 RAY DIFFERENTIALS FOR SPECULAR REFLECTION AND TRANSMISSION

Given the effectiveness of ray differentials for finding filter regions for texture antialiasing for camera rays, it is useful to extend the method to make it possible to determine texture space sampling rates for objects that are seen indirectly via specular reflection or refraction; objects seen in mirrors, for example, should also no longer have texture aliasing than directly visible objects. Igehy (1999) developed an elegant solution to the problem of how to find the appropriate differential rays for specular reflection and refraction, which is the approach used in pbrt.²

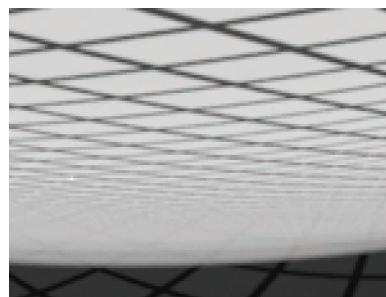
² Igehy's formulation is slightly different from the one here—he effectively tracked the differences between the main ray and the offset rays, while we store the offset rays explicitly. The mathematics all work out to be the same in the end; we chose this alternative because we believe that it makes the algorithm's operation for camera rays easier to understand.



(a)



(b)



(c)

Figure 10.5: (a) Tracking ray differentials for reflected and refracted rays ensures that the image map texture seen in the balls is filtered to avoid aliasing. The left ball is glass, exhibiting reflection and refraction, and the right ball is a mirror, just showing reflection. Note that the texture is well filtered over both of the balls. (b) and (c) show a zoomed-in section of the glass ball; (b) shows the aliasing artifacts that are present if ray differentials aren't used, while (c) shows the result when they are.

Figure 10.5 illustrates the difference that proper texture filtering for specular reflection and transmission can make. Figure 10.5(a) shows a glass ball and a mirrored ball on a plane with a texture map containing high-frequency components. Ray differentials ensure that the images of the texture seen via reflection and refraction from the balls are free of aliasing artifacts. A close-up view of the reflection in the glass ball is shown in Figure 10.5(b) and (c); Figure 10.5(b) was rendered without ray differentials for the reflected and transmitted rays, and Figure 10.5(c) was rendered with ray differentials. (All images are rendered with one sample per pixel.) The aliasing errors in the left image are eliminated on the right without excessively blurring the texture.

In order to compute the reflected or transmitted ray differentials at a surface intersection point, we need an approximation to the rays that would have been traced at the intersection points for the two offset rays in the ray differential that hit the surface (Figure 10.6).

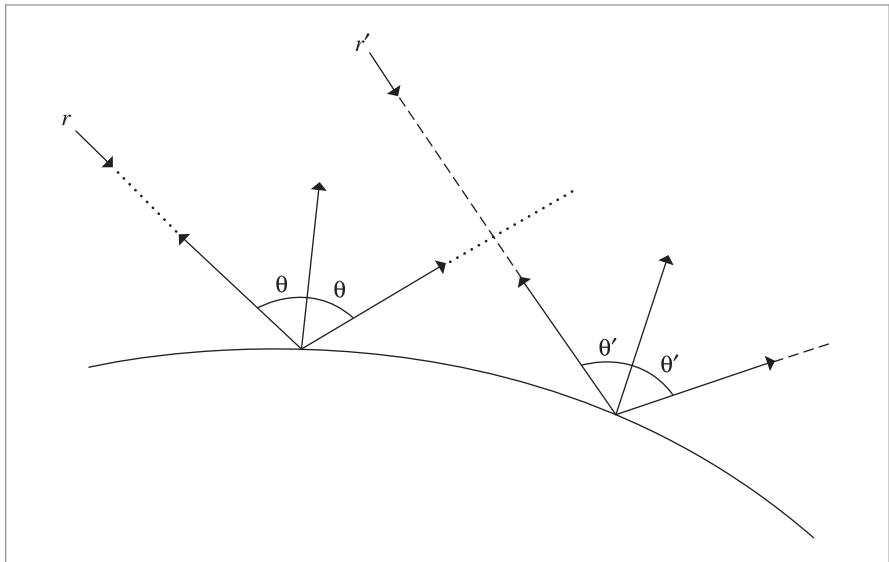


Figure 10.6: The specular reflection formula gives the direction of the reflected ray at a point on a surface. An offset ray for a ray differential r' (dashed line) will generally intersect the surface at a different point and be reflected in a different direction. The new direction is affected by both the different surface normal at the point as well as the offset ray's different incident direction. The computation to find the reflected direction for the offset ray in pbrt estimates the change in reflected direction as a function of image space position and approximates the ray differential's direction with the main ray's direction added to the estimated change in direction.

The new ray for the main ray is computed by the BSDF, so here we only need to compute the outgoing rays for the r_x and r_y differentials.

For both reflection and refraction, the origin of each differential ray is easily found. The `SurfaceInteraction::ComputeDifferentials()` method previously computed approximations for how much the surface position changes with respect to (x, y) position on the image plane $\partial p / \partial x$ and $\partial p / \partial y$. Adding these offsets to the intersection point of the main ray gives approximate origins for the new rays. If the incident ray doesn't have differentials, then it's impossible to compute reflected ray differentials and this step is skipped.

```

<Compute ray differential rd for specular reflection> ≡
    RayDifferential rd = isect.SpawnRay(wi);
    if (ray.hasDifferentials) {
        rd.hasDifferentials = true;
        rd.rxOrigin = isect.p + isect.dpdx;
        rd.ryOrigin = isect.p + isect.dpdy;
        <Compute differential reflected directions 608>
    }

```

38

```

Interaction::p 115
Interaction::SpawnRay() 232
RayDifferential 75
RayDifferential::  
hasDifferentials  
75

```

```

SurfaceInteraction::  
ComputeDifferentials()  
601

```

```

SurfaceInteraction::dpdx 600
SurfaceInteraction::dpdy 600

```

Finding the directions of these rays is slightly trickier. Igehy (1999) observed that if we know how much the reflected direction ω_i changes with respect to a shift of a pixel sample in the x and y directions on the image plane, we can use this information to approximate

the direction of the offset rays. For example, the direction for the ray offset in x is

$$\omega \approx \omega_i + \frac{\partial \omega_i}{\partial x}.$$

Recall from Equation (8.6) that for a general world space surface normal and outgoing direction, the direction for perfect specular reflection is

$$\omega_i = -\omega_o + 2(\omega_o \cdot \mathbf{n})\mathbf{n}.$$

Fortunately, the partial derivatives of this expression are easily computed:

$$\begin{aligned}\frac{\partial \omega_i}{\partial x} &= \frac{\partial}{\partial x} (-\omega_o + 2(\omega_o \cdot \mathbf{n})\mathbf{n}) \\ &= -\frac{\partial \omega_o}{\partial x} + 2 \left((\omega_o \cdot \mathbf{n}) \frac{\partial \mathbf{n}}{\partial x} + \frac{\partial(\omega_o \cdot \mathbf{n})}{\partial x} \mathbf{n} \right).\end{aligned}$$

Using the properties of the dot product, it can be shown that

$$\frac{\partial(\omega_o \cdot \mathbf{n})}{\partial x} = \frac{\partial \omega_o}{\partial x} \cdot \mathbf{n} + \omega_o \cdot \frac{\partial \mathbf{n}}{\partial x}.$$

The value of $\partial \omega_o / \partial x$ can be found from the difference between the direction of the ray differential's main ray and the direction of the r_x offset ray, and all of the other necessary quantities are readily available from the SurfaceInteraction, so the implementation of this computation for the partial derivatives in x and y is straightforward.

(Compute differential reflected directions) ≡

607

```
Normal3f dndx = isect.shading.dndu * isect.dudx +
                  isect.shading.dndv * isect.dvdx;
Normal3f dndy = isect.shading.dndu * isect.dudy +
                  isect.shading.dndv * isect.dvdy;
Vector3f dwodx = -ray.rxDirection - wo, dwody = -ray.ryDirection - wo;
Float dDNDx = Dot(dwodx, ns) + Dot(wo, dndx);
Float dDNDy = Dot(dwody, ns) + Dot(wo, dndy);
rd.rxDirection = wi - dwodx +
  2.f * Vector3f(Dot(wo, ns) * dndx + dDNDx * ns);
rd.ryDirection = wi - dwody +
  2.f * Vector3f(Dot(wo, ns) * dndy + dDNDy * ns);
```

Float 1062

Normal3f 71

RayDifferential::rxDirection
75

RayDifferential::ryDirection
75

SurfaceInteraction::dudx 600

SurfaceInteraction::dudy 600

SurfaceInteraction::dvdx 600

SurfaceInteraction::dvdy 600

SurfaceInteraction::shading
118

SurfaceInteraction::
shading::dndu
118

SurfaceInteraction::
shading::dndv
118

Vector3f 60

A similar process of differentiating the equation for the direction of a specularly transmitted ray, Equation (8.9), gives the equation to find the differential change in the transmitted direction. We won't include the derivation or our implementation here, but refer the interested reader to the original paper and to the pbrt source code, respectively.

10.2 TEXTURE COORDINATE GENERATION

Almost all of the textures in this chapter are functions that take a 2D or 3D coordinate and return a texture value. Sometimes there are obvious ways to choose these texture coordinates; for parametric surfaces, such as the quadrics in Chapter 3, there is a natural 2D (u, v) parameterization of the surface, and for all surfaces the shading point p is a natural choice for a 3D coordinate.

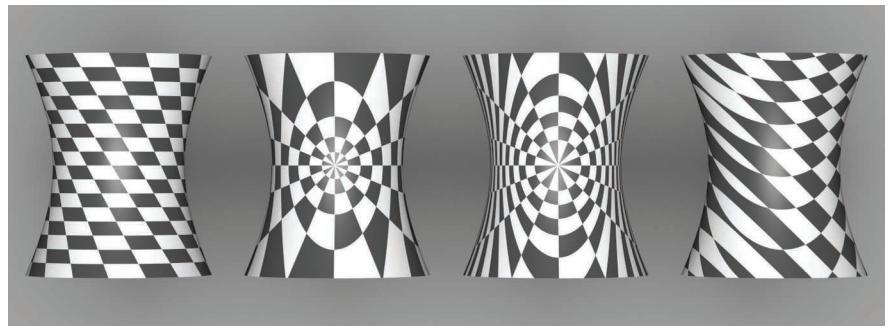


Figure 10.7: A checkerboard texture, applied to a hyperboloid with different texture coordinate generation techniques. From left to right, (u, v) mapping, spherical mapping, cylindrical mapping, and planar mapping.

In other cases, there is no natural parameterization, or the natural parameterization may be undesirable. For instance, the (u, v) values near the poles of spheres are severely distorted. Also, for an arbitrary subdivision surface, there is no simple, general-purpose way to assign texture values so that the entire $[0, 1]^2$ space is covered continuously and without distortion. In fact, creating smooth parameterizations of complex meshes with low distortion is an active area of research in computer graphics.

This section starts by introducing two abstract base classes—`TextureMapping2D` and `TextureMapping3D`—that provide an interface for computing these 2D and 3D texture coordinates. We will then implement a number of standard mappings using this interface (Figure 10.7 shows a number of them). Texture implementations store a pointer to a 2D or 3D mapping function as appropriate and use it to compute the texture coordinates at each point. Thus, it’s easy to add new mappings to the system without having to modify all of the Texture implementations, and different mappings can be used for different textures associated with the same surface. In pbrt, we will use the convention that 2D texture coordinates are denoted by (s, t) ; this helps make clear the distinction between the intrinsic (u, v) parameterization of the underlying surface and the (possibly different) coordinate values used for texturing.

The `TextureMapping2D` base class has a single method, `TextureMapping2D::Map()`, which is given the `SurfaceInteraction` at the shading point and returns the (s, t) texture coordinates via a `Point2f`. It also returns estimates for the change in (s, t) with respect to pixel x and y coordinates in the `dstdx` and `dstdy` parameters so that textures that use the mapping can determine the (s, t) sampling rate and filter accordingly.

`SurfaceInteraction` 116

`Texture` 614

`TextureMapping2D` 609

`TextureMapping2D::Map()` 610

`TextureMapping3D` 614

(Texture Declarations) ≡

```
class TextureMapping2D {
public:
    (TextureMapping2D Interface 610)
};
```

```
<TextureMapping2D Interface> ≡ 609
    virtual Point2f Map(const SurfaceInteraction &si,
                        Vector2f *dstdx, Vector2f *dstdy) const = 0;
```

10.2.1 2D (u, v) MAPPING

The simplest mapping uses the (u, v) coordinates in the `SurfaceInteraction` to compute the texture coordinates. Their values can be offset and scaled with user-supplied values in each dimension.

```
<Texture Declarations> +≡
class UVMapping2D : public TextureMapping2D {
public:
    (UVMapping2D Public Methods)
private:
    const Float su, sv, du, dv;
};
```

```
<Texture Method Definitions> ≡
UVMapping2D::UVMapping2D(Float su, Float sv, Float du, Float dv)
    : su(su), sv(sv), du(du), dv(dv) { }
```

The scale-and-shift computation to compute (s, t) coordinates is straightforward:

```
<Texture Method Definitions> +≡
Point2f UVMapping2D::Map(const SurfaceInteraction &si,
                         Vector2f *dstdx, Vector2f *dstdy) const {
    (Compute texture differentials for 2D (u, v) mapping 611)
    return Point2f(si.uv[0] * su + du,
                  si.uv[1] * sv + dv);
}
```

Computing the differential change in s and t in terms of the original change in u and v and the scale amounts is also easy. Using the chain rule,

$$\frac{\partial s}{\partial x} = \frac{\partial u}{\partial x} \frac{\partial s}{\partial u} + \frac{\partial v}{\partial x} \frac{\partial s}{\partial v}$$

and similarly for the three other partial derivatives. From the mapping method,

$$s = s_u u + d_u,$$

so

$$\frac{\partial s}{\partial u} = s_u, \quad \frac{\partial s}{\partial v} = 0,$$

and thus

$$\frac{\partial s}{\partial x} = s_u \frac{\partial u}{\partial x},$$

and so forth.

Float 1062
 Point2f 68
 SurfaceInteraction 116
 SurfaceInteraction::uv 116
 TextureMapping2D 609
 UVMapping2D 610
 UVMapping2D::du 610
 UVMapping2D::dv 610
 UVMapping2D::su 610
 UVMapping2D::sv 610
 Vector2f 60

```
(Compute texture differentials for 2D (u, v) mapping) +≡
    *dstdx = Vector2f(su * si.dudx, sv * si.dvdx);
    *dstdy = Vector2f(su * si.dudy, sv * si.dvdy);
```

610

10.2.2 SPHERICAL MAPPING

Another useful mapping effectively wraps a sphere around the object. Each point is projected along the vector from the sphere's center through the point, up to the sphere's surface. There, the (u, v) mapping for the sphere shape is used. The `SphericalMapping2D` stores a transformation that is applied to points before this mapping is performed; this effectively allows the mapping sphere to be arbitrarily positioned and oriented with respect to the object.

```
(Texture Declarations) +≡
class SphericalMapping2D : public TextureMapping2D {
public:
    (SphericalMapping2D Public Methods)
private:
    Point2f sphere(const Point3f &p) const;
    const Transform WorldToTexture;
};

(Texture Method Definitions) +≡
Point2f SphericalMapping2D::Map(const SurfaceInteraction &si,
    Vector2f *dstdx, Vector2f *dstdy) const {
    Point2f st = sphere(si.p);
    (Compute texture coordinate differentials for sphere (u, v) mapping 612)
    (Handle sphere mapping discontinuity for coordinate differentials 612)
    return st;
}
```

Float 1062
 Interaction::: 115
 Inv2Pi 1063
 InvPi 1063
 Point2f 68
 Point3f 68
 SphericalMapping2D 611
 SphericalMapping2D::sphere() 611
 SphericalMapping2D:::
 WorldToTexture 611
 SphericalPhi() 346
 SphericalTheta() 346
 SurfaceInteraction 116
 TextureMapping2D 609
 Transform 83
 UVMapping2D::su 610
 UVMapping2D::sv 610
 Vector2f 60
 Vector3f::Normalize() 66
 Vector3f 60

A short utility function computes the mapping for a single point. It will be useful to have this logic separated out for computing texture coordinate differentials.

```
(Texture Method Definitions) +≡
Point2f SphericalMapping2D::sphere(const Point3f &p) const {
    Vector3f vec = Normalize(WorldToTexture(p) - Point3f(0,0,0));
    Float theta = SphericalTheta(vec), phi = SphericalPhi(vec);
    return Point2f(theta * InvPi, phi * Inv2Pi);
}
```

We could use the chain rule again to compute the texture coordinate differentials but will instead use a forward differencing approximation to demonstrate another way to compute these values that is useful for more complex mapping functions. Recall that the `SurfaceInteraction` stores the screen space partial derivatives $\partial p / \partial x$ and $\partial p / \partial y$ that give the change in position as a function of change in image sample position. Therefore, if the s coordinate is computed by some function $f_s(p)$, it's easy to compute approximations like

$$\frac{\partial s}{\partial x} \approx \frac{f_s(p + \Delta \partial p / \partial x) - f_s(p)}{\Delta}.$$

As the distance Δ approaches 0, this gives the actual partial derivative at p .

(Compute texture coordinate differentials for sphere (u, v) mapping) \equiv 611

```
const Float delta = .1f;
Point2f stDeltaX = sphere(si.p + delta * si.dpdx);
*dstdx = (stDeltaX - st) / delta;
Point2f stDeltaY = sphere(si.p + delta * si.dpdy);
*dstdy = (stDeltaY - st) / delta;
```

One other detail is that the sphere mapping has a discontinuity in the mapping formula; there is a seam at $t = 1$, where the t texture coordinate discontinuously jumps back to zero. We can detect this case by checking to see if the absolute value of the estimated derivative computed with forward differencing is greater than 0.5 and then adjusting it appropriately.

(Handle sphere mapping discontinuity for coordinate differentials) \equiv 611

```
if ((*dstdx)[1] > .5) (*dstdx)[1] = 1 - (*dstdx)[1];
else if ((*dstdx)[1] < -.5f) (*dstdx)[1] = -((*dstdx)[1] + 1);
if ((*dstdy)[1] > .5) (*dstdy)[1] = 1 - (*dstdy)[1];
else if ((*dstdy)[1] < -.5f) (*dstdy)[1] = -((*dstdy)[1] + 1);
```

10.2.3 CYLINDRICAL MAPPING

The cylindrical mapping effectively wraps a cylinder around the object. It also supports a transformation to orient the mapping cylinder.

(Texture Declarations) \equiv

```
class CylindricalMapping2D : public TextureMapping2D {
public:
    (CylindricalMapping2D Public Methods)
private:
    (CylindricalMapping2D Private Methods 613)
    const Transform WorldToTexture;
};
```

The cylindrical mapping has the same basic structure as the sphere mapping; just the mapping function is different. Therefore, we will omit the fragment that computes texture coordinate differentials, since it is essentially the same as the spherical version.

(Texture Method Definitions) \equiv

```
Point2f CylindricalMapping2D::Map(const SurfaceInteraction &si,
    Vector2f *dstdx, Vector2f *dstdy) const {
    Point2f st = cylinder(si.p);
    (Compute texture coordinate differentials for cylinder ( $u, v$ ) mapping)
    return st;
}
```

CylindricalMapping2D 612
 CylindricalMapping2D::
 cylinder() 613
 Float 1062
 Interaction::p 115
 Point2f 68
 SurfaceInteraction 116
 SurfaceInteraction::dpdx 600
 SurfaceInteraction::dpdy 600
 TextureMapping2D 609
 Transform 83
 Vector2f 60

```
<CylindricalMapping2D Private Methods> ≡ 612
    Point2f cylinder(const Point3f &p) const {
        Vector3f vec = Normalize(WorldToTexture(p) - Point3f(0,0,0));
        return Point2f((Pi + std::atan2(vec.y, vec.x)) * Inv2Pi,
                      vec.z);
    }
```

10.2.4 PLANAR MAPPING

Another classic mapping method is planar mapping. The point is effectively projected onto a plane; a 2D parameterization of the plane then gives texture coordinates for the point. For example, a point p might be projected onto the $z = 0$ plane to yield texture coordinates given by $s = p_x$ and $t = p_y$.

In general, we can define such a parameterized plane with two nonparallel vectors \mathbf{v}_s and \mathbf{v}_t and offsets d_s and d_t . The texture coordinates are given by the coordinates of the point with respect to the plane's coordinate system, which are computed by taking the dot product of the vector from the point to the origin with each vector \mathbf{v}_s and \mathbf{v}_t and then adding the offset. For the example in the previous paragraph, we'd have $\mathbf{v}_s = (1, 0, 0)$, $\mathbf{v}_t = (0, 1, 0)$, and $d_s = d_t = 0$.

```
<Texture Declarations> +≡
    class PlanarMapping2D : public TextureMapping2D {
public:
    <PlanarMapping2D Public Methods 613>
private:
    const Vector3f vs, vt;
    const Float ds, dt;
};

<PlanarMapping2D Public Methods> ≡ 613
    PlanarMapping2D(const Vector3f &vs, const Vector3f &vt,
                    Float ds = 0, Float dt = 0)
        : vs(vs), vt(vt), ds(ds), dt(dt) { }
```

CylindricalMapping2D:
WorldToTexture
612
Dot() 63
Float 1062
Interaction::p 115
Inv2Pi 1063
Pi 1063
PlanarMapping2D 613
PlanarMapping2D::ds 613
PlanarMapping2D::dt 613
PlanarMapping2D::vs 613
PlanarMapping2D::vt 613
Point2f 68
Point3f 68
SurfaceInteraction 116
SurfaceInteraction::dpdx 600
SurfaceInteraction::dpdy 600
TextureMapping2D 609
TextureMapping3D 614
Vector2f 60
Vector3f::Normalize() 66
Vector3f 60

The planar mapping differentials can be computed directly by finding the differentials of the point p in texture coordinate space.

```
<Texture Method Definitions> +≡
    Point2f PlanarMapping2D::Map(const SurfaceInteraction &si,
                                 Vector2f *dstdx, Vector2f *dstdy) const {
        Vector3f vec(si.p);
        *dstdx = Vector2f(Dot(si.dpdx, vs), Dot(si.dpdx, vt));
        *dstdy = Vector2f(Dot(si.dpdy, vs), Dot(si.dpdy, vt));
        return Point2f(ds + Dot(vec, vs), dt + Dot(vec, vt));
    }
```

10.2.5 3D MAPPING

We will also define a `TextureMapping3D` class that defines the interface for generating 3D texture coordinates.

```
(Texture Declarations) +≡
class TextureMapping3D {
public:
    (TextureMapping3D Interface 614)
};

(TextureMapping3D Interface) ≡ 614
virtual Point3f Map(const SurfaceInteraction &si,
    Vector3f *dpdx, Vector3f *dpdy) const = 0;
```

The natural 3D mapping just takes the world space coordinate of the point and applies a linear transformation to it. This will often be a transformation that takes the point back to the primitive's object space.

```
(Texture Declarations) +≡
class TransformMapping3D : public TextureMapping3D {
public:
    (TransformMapping3D Public Methods)
private:
    const Transform WorldToTexture;
};
```

Because a linear mapping is used, the differential change in texture coordinates can be found by applying the same mapping to the partial derivatives of position.

```
(Texture Method Definitions) +≡
Point3f TransformMapping3D::Map(const SurfaceInteraction &si,
    Vector3f *dpdx, Vector3f *dpdy) const {
    *dpdx = WorldToTexture(si.dpdx);
    *dpdy = WorldToTexture(si.dpdy);
    return WorldToTexture(si.p);
}
```

10.3 TEXTURE INTERFACE AND BASIC TEXTURES

Texture is a template class parameterized by the return type of its evaluation function. This design makes it possible to reuse almost all of the code among textures that return different types. pbrt currently uses only `Float` and `Spectrum` textures.

```
(Texture Declarations) +≡
template <typename T> class Texture {
public:
    (Texture Interface 615)
};
```

The key to `Texture`'s interface is its evaluation function; it returns a value of the template type `T`. The only information it has access to in order to evaluate its value is the `SurfaceInteraction` at the point being shaded. Different textures in this chapter will use different parts of this structure to drive their evaluation.

- Interaction::[115](#)
- Point3f [68](#)
- Spectrum [315](#)
- SurfaceInteraction [116](#)
- SurfaceInteraction::dpdx [600](#)
- Texture [614](#)
- TextureMapping3D [614](#)
- Transform [83](#)
- TransformMapping3D [614](#)
- TransformMapping3D::WorldToTexture [614](#)
- Vector3f [60](#)

(Texture Interface) ≡

```
virtual T Evaluate(const SurfaceInteraction &) const = 0;
```

614

10.3.1 CONSTANT TEXTURE

ConstantTexture returns the same value no matter where it is evaluated. Because it represents a constant function, it can be accurately reconstructed with any sampling rate and therefore needs no antialiasing. Although this texture is trivial, it is actually quite useful. By providing this class, all parameters to all Materials can be represented as Textures, whether they are spatially varying or not. For example, a red diffuse object will have a ConstantTexture that always returns red as the diffuse color of the material. This way, the shading system always evaluates a texture to get the surface properties at a point, avoiding the need for separate textured and nontextured versions of materials. This material's implementation is in the files `textures/constant.h` and `textures/constant.cpp`.

(ConstantTexture Declarations) ≡

```
template <typename T> class ConstantTexture : public Texture<T> {
public:
    (ConstantTexture Public Methods 615)
private:
    T value;
};
```

(ConstantTexture Public Methods) ≡

```
ConstantTexture(const T &value) : value(value) { }
T Evaluate(const SurfaceInteraction &) const {
    return value;
}
```

615

10.3.2 SCALE TEXTURE

We have defined the texture interface in a way that makes it easy to use the output of one texture function when computing another. This is useful since it lets us define generic texture operations using any of the other texture types. The ScaleTexture takes two textures and returns the product of their values when evaluated. It is defined in `textures/scale.h` and `textures/scale.cpp`.

(ScaleTexture Declarations) ≡

```
template <typename T1, typename T2>
class ScaleTexture : public Texture<T2> {
public:
    (ScaleTexture Public Methods 616)
private:
    (ScaleTexture Private Data 616)
};
```

ConstantTexture 615
Material 577
ScaleTexture 615
SurfaceInteraction 116
Texture 614

The attentive reader may notice that the `shared_ptr` parameters to the constructor are stored in member variables and wonder if there is a performance issue from this approach along the lines of the one described in Section 9.3 with regard to the `Bump()` method. In this case we're fine: Textures are only created at scene creation time, rather than at

rendering time for each ray. Therefore, there are no issues with contention at the memory location that stores the reference count for each `shared_ptr`.

(ScaleTexture Public Methods) ≡

```
ScaleTexture(const std::shared_ptr<Texture<T1>> &tex1,
             const std::shared_ptr<Texture<T2>> &tex2)
    : tex1(tex1), tex2(tex2) { }
```

615

Note that the return types of the two textures can be different; the implementation here just requires that it be possible to multiply their values together. Thus, a `Float` texture can be used to scale a `Spectrum` texture.

(ScaleTexture Private Data) ≡

```
std::shared_ptr<Texture<T1>> tex1;
std::shared_ptr<Texture<T2>> tex2;
```

615

`ScaleTexture` ignores antialiasing, leaving it to its two subtextures to antialias themselves but not making an effort to antialias their product. While it is easy to show that the product of two band-limited functions is also band limited, the maximum frequency present in the product may be greater than that of either of the two terms individually. Thus, even if the scale and value textures are perfectly antialiased, the result might not be. Fortunately, the most common use of this texture is to scale another texture by a constant, in which case the other texture's antialiasing is sufficient.

(ScaleTexture Public Methods) +≡

```
T2 Evaluate(const SurfaceInteraction &si) const {
    return tex1->Evaluate(si) * tex2->Evaluate(si);
}
```

615

10.3.3 MIX TEXTURES

The `MixTexture` class is a more general variation of `ScaleTexture`. It takes three textures as input: two may be of any single type, and the third must return a floating-point value. The floating-point texture is then used to linearly interpolate between the two other textures. Note that a `ConstantTexture` could be used for the floating-point value to achieve a uniform blend, or a more complex `Texture` could be used to blend in a spatially nonuniform way. This texture is defined in `textures/mix.h` and `textures/mix.cpp`.

(MixTexture Declarations) ≡

```
template <typename T> class MixTexture : public Texture<T> {
public:
    (MixTexture Public Methods 616)
private:
    std::shared_ptr<Texture<T>> tex1, tex2;
    std::shared_ptr<Texture<Float>> amount;
};
```

(MixTexture Public Methods) ≡

```
MixTexture(const std::shared_ptr<Texture<T>> &tex1,
           const std::shared_ptr<Texture<T>> &tex2,
           const std::shared_ptr<Texture<Float>> &amount)
    : tex1(tex1), tex2(tex2), amount(amount) { }
```

616

ConstantTexture 615
`Float` 1062
`MixTexture` 616
`MixTexture::amount` 616
`MixTexture::tex1` 616
`MixTexture::tex2` 616
`ScaleTexture` 615
`ScaleTexture::tex1` 616
`ScaleTexture::tex2` 616
`SurfaceInteraction` 116
`Texture` 614
`Texture::Evaluate()` 615

To evaluate the mixture, the three textures are evaluated and the floating-point value is used to linearly interpolate between the two. When the blend amount `amt` is zero, the first texture's value is returned, and when it is one the second one's value is returned. We will generally assume that `amt` will be between zero and one, but this behavior is not enforced, so extrapolation is possible as well. As with the `ScaleTexture`, antialiasing is ignored, so the introduction of aliasing here is a possibility.

```
<MixTexture Public Methods> +≡ 616
T Evaluate(const SurfaceInteraction &si) const {
    T t1 = tex1->Evaluate(si), t2 = tex2->Evaluate(si);
    Float amt = amount->Evaluate(si);
    return (1 - amt) * t1 + amt * t2;
}
```

10.3.4 BILINEAR INTERPOLATION

```
<BilerpTexture Declarations> ≡
template <typename T> class BilerpTexture : public Texture<T> {
public:
    <BilerpTexture Public Methods 617>
private:
    <BilerpTexture Private Data 617>
};
```

The `BilerpTexture` class provides bilinear interpolation among four constant values. Values are defined at $(0, 0)$, $(1, 0)$, $(0, 1)$, and $(1, 1)$ in (s, t) parameter space. The value at a particular (s, t) position is found by interpolating between them. It is defined in the files `textures/bilerp.h` and `textures/bilerp.cpp`.

```
<BilerpTexture Public Methods> ≡ 617
BilerpTexture(std::unique_ptr<TextureMapping2D> mapping, const T &v00,
              const T &v01, const T &v10, const T &v11)
: mapping(std::move(mapping)), v00(v00), v01(v01), v10(v10),
  v11(v11) { }
```

```
<BilerpTexture Private Data> ≡ 617
std::unique_ptr<TextureMapping2D> mapping;
const T v00, v01, v10, v11;
```

The interpolated value of the four values at an (s, t) position can be computed by three linear interpolations. For example, we can first use s to interpolate between the values at $(0, 0)$ and $(1, 0)$ and store that in a temporary `tmp1`. We can then do the same for the $(0, 1)$ and $(1, 1)$ values and store the result in `tmp2`. Finally, we use t to interpolate between `tmp1` and `tmp2` and obtain the final result. Mathematically, this is

$$\begin{aligned} \text{tmp}_1 &= (1 - s)v_{00} + s v_{10} \\ \text{tmp}_2 &= (1 - s)v_{01} + s v_{11} \\ \text{result} &= (1 - t)\text{tmp}_1 + t \text{tmp}_2. \end{aligned}$$

BilerpTexture 617
BilerpTexture::mapping 617
BilerpTexture::v00 617
BilerpTexture::v01 617
BilerpTexture::v10 617
BilerpTexture::v11 617
Float 1062
ScaleTexture 615
SurfaceInteraction 116
Texture 614
Texture::Evaluate() 615
TextureMapping2D 609

Rather than storing the intermediate values explicitly, we can perform some algebraic rearrangement to give us the same result from an appropriately weighted average of the

four corner values:

$$\text{result} = (1-s)(1-t)v_{00} + (1-s)t v_{01} + s(1-t)v_{10} + s t v_{11}.$$

(BilerpTexture Public Methods) \equiv

617

```
T Evaluate(const SurfaceInteraction &si) const {
    Vector2f dstdx, dstdy;
    Point2f st = mapping->Map(si, &dstdx, &dstdy);
    return (1-st[0]) * (1-st[1]) * v00 + (1-st[0]) * (st[1]) * v01 +
        (st[0]) * (1-st[1]) * v10 + (st[0]) * (st[1]) * v11;
}
```

10.4 IMAGE TEXTURE

The `ImageTexture` class stores a 2D array of point-sampled values of a texture function. It uses these samples to reconstruct a continuous image function that can be evaluated at an arbitrary (s, t) position. These sample values are often called *texels*, since they are similar to pixels in an image but are used in the context of a texture. Image textures are the most widely used type of texture in computer graphics; digital photographs, scanned artwork, images created with image-editing programs, and images generated by renderers are all extremely useful sources of data for this particular texture representation (Figure 10.8). The term *texture map* is often used to refer to this type of texture, although this usage blurs the distinction between the mapping that computes texture



Figure 10.8: An Example of Image Textures. Image textures are used throughout the San Miguel scene to represent spatially varying surface appearance properties. (left) Scene rendered with image textures. (right) Each image texture has been replaced with its average value. Note how much visual richness is lost.

BilerpTexture::mapping 617

BilerpTexture::v00 617

BilerpTexture::v01 617

BilerpTexture::v10 617

BilerpTexture::v11 617

Point2f 68

SurfaceInteraction 116

TextureMapping2D::Map() 610

Vector2f 60

coordinates and the texture function itself. The implementation of this texture is in the files `textures/imagemap.h` and `textures/imagemap.cpp`.

The `ImageTexture` class is different from other textures in the system in that it is parameterized on both the data type of the texels it stores in memory as well as the data type of the value that it returns. Making this distinction allows us to create, for example, `ImageTextures` that store `RGBSpectrum` values in memory, but always return `Spectrum` values. In this way, when the system is compiled with full-spectral rendering enabled, the memory cost to store full `SampledSpectrum` texels doesn't need to be paid for source images that only have RGB components.

```
(ImageTexture Declarations) ≡
    template <typename Tmemory, typename Treturn>
        class ImageTexture : public Texture<Treturn> {
    public:
        (ImageTexture Public Methods 622)
    private:
        (ImageTexture Private Methods 622)
        (ImageTexture Private Data 619)
    };
```

The caller provides the `ImageTexture` with the filename of an image map, parameters that control the filtering of the map for antialiasing, and parameters that make it possible to scale and gamma-correct the texture values. The scale parameter will be explained later in this section, and the texture filtering parameters will be explained in Section 10.4.3. The contents of the file are used to create an instance of the `MIPMap` class that stores the texels in memory and handles the details of reconstruction and filtering to reduce aliasing.

For an `ImageTexture` that stores `RGBSpectrum` values in memory, its `MIPMap` stores the image data using three floating-point values for each sample. This is a somewhat wasteful representation, since a single image map may have millions of texels and may not need the full 32 bits of accuracy from the `Floats` used to store RGB values for each of them. Exercise 10.1 at the end of this chapter discusses this issue further.

```
(ImageTexture Method Definitions) ≡
    template <typename Tmemory, typename Treturn>
    ImageTexture<Tmemory, Treturn>::ImageTexture(
        std::unique_ptr<TextureMapping2D> mapping,
        const std::string &filename, bool doTrilinear, Float maxAniso,
        ImageWrap wrapMode, Float scale, bool gamma)
        : mapping(std::move(mapping)) {
            mipmap = GetTexture(filename, doTrilinear, maxAniso,
                wrapMode, scale, gamma);
    }

(ImageTexture Private Data) ≡
    std::unique_ptr<TextureMapping2D> mapping;
    MIPMap<Tmemory> *mipmap;
```

10.4.1 TEXTURE MEMORY MANAGEMENT

Each image map may require a meaningful amount of memory, and a complex scene may have thousands of image maps. Because each image texture may be reused many times within a scene, pbrt maintains a table of image maps that have been loaded so far, so that they are only loaded into memory once even if they are used in more than one `ImageTexture`. The `ImageTexture` constructor calls the static `ImageTexture::GetTexture()` method to get a `MIPMap` representation of the desired texture.

(ImageTexture Method Definitions) +≡

```
template <typename Tmemory, typename Treturn> MIPMap<Tmemory> *
ImageTexture<Tmemory, Treturn>::GetTexture(const std::string &filename,
    bool doTrilinear, Float maxAniso, ImageWrap wrap, Float scale,
    bool gamma) {
    (Return MIPMap from texture cache if present 620)
    (Create MIPMap for filename 620)
    return mipmap;
}
```

`TexInfo` is a simple structure that holds the image map's filename and filtering parameters; all of these must match for a `MIPMap` to be reused in another `ImageTexture`. Its definition is straightforward (its members exactly correspond to the parameters of the `GetTexture()` method) and won't be included here.

(ImageTexture Private Data) +≡

619

```
static std::map<TexInfo, std::unique_ptr<MIPMap<Tmemory>>> textures;
```

(Return MIPMap from texture cache if present) ≡

620

```
TexInfo texInfo(filename, doTrilinear, maxAniso, wrap, scale, gamma);
if (textures.find(texInfo) != textures.end())
    return textures[texInfo].get();
```

If the texture hasn't been loaded yet, a call to `ReadImage()` yields the image contents.

(Create MIPMap for filename) ≡

620

```
Point2i resolution;
std::unique_ptr<RGBSpectrum[]> texels = ReadImage(filename, &resolution);
MIPMap<Tmemory> *mipmap = nullptr;
if (texels) {
    (Convert texels to type Tmemory and create MIPMap 621)
} else {
    (Create one-valued MIPMap 622)
}
textures[texInfo].reset(mipmap);
```

Float 1062

ImageTexture 619

ImageTexture::convertIn()
622

ImageTexture::GetTexture()
620

ImageTexture::textures 620

ImageWrap 626

MIPMap 625

Point2i 68

ReadImage() 1067

RGBSpectrum 332

TexInfo 620

Because `ReadImage()` returns an array of `RGBSpectrum` values for the texels, it is necessary to convert these values to the particular type `Tmemory` of texel that this `MIPMap` is storing (e.g., `Float`) if the type of `Tmemory` isn't `RGBSpectrum`. The per-texel conversion is handled by the utility routine `ImageTexture::convertIn()`.

```
<Convert texels to type Tmemory and create MIPMap> ≡ 620
    std::unique_ptr<Tmemory[]> convertedTexels(new Tmemory[resolution.x *
                                                resolution.y]);
    for (int i = 0; i < resolution.x * resolution.y; ++i)
        convertIn(texels[i], &convertedTexels[i], scale, gamma);
    mipmap = new MIPMap<Tmemory>(resolution, convertedTexels.get(),
                                   doTrilinear, maxAniso, wrap);
```

Per-texel conversion is done using C++ function overloading. For every type to which we would like to be able to convert these values, a separate `ImageTexture::convertIn()` function must be provided. In the loop over texels earlier, C++’s function overloading mechanism will select the appropriate instance of `ImageTexture::convertIn()` based on the destination type. Unfortunately, it is not possible to return the converted value from the function, since C++ doesn’t support overloading by return type.

In addition to converting types, these functions optionally scale and gamma correct the texel values to map them to a desired range. Gamma correction is particularly important to handle carefully: computer displays generally don’t exhibit a linear relationship between the pixel values to be displayed and the radiance that they emit. Thus, an artist may create a texture map where, as seen on an LCD display, one part of the image appears twice as bright as another. However, the corresponding pixel values won’t in fact have a 2:1 relationship. (Conversely, pixels whose values have a 2:1 relationship don’t lead to a 2:1 brightness ratio.) This discrepancy is a problem for a renderer using such an image as a texture map, since the renderer usually expects a linear relationship between texel values and the quantity that they represent.

`pbrt` follows the sRGB standard, which prescribes a specific transfer curve matching the typical behavior of CRT displays. This standard is widely supported on 2015-era devices; other (non-CRT) devices such as LCDs or inkjet printers typically accept sRGB gamma-corrected values as input and then re-map them internally to match the device-specific behavior.

The sRGB gamma curve is a piecewise function with a linear term for low values and a power law for medium to large values.

$$\gamma(x) = \begin{cases} 12.92x, & x \leq 0.0031308 \\ (1.055)x^{1/2.4} - 0.055, & x > 0.0031308 \end{cases}$$

```
<Global Inline Functions> +≡
    inline Float GammaCorrect(Float value) {
        if (value <= 0.0031308f)
            return 12.92f * value;
        return 1.055f * std::pow(value, (Float)(1.f / 2.4f)) - 0.055f;
    }
```

Float 1062
 GammaCorrect() 621
 ImageTexture::convertIn()
 622
 MIPMap 625
 WriteImage() 1068

The `GammaCorrect()` will be used to write sRGB-compatible 8-bit image files in the `WriteImage()` function. To import textures into `pbrt`, we are interested in the opposite direction: removing an existing gamma correction to reestablish a linear relationship between brightness and pixel values.

```
(Global Inline Functions) +≡
    inline Float InverseGammaCorrect(Float value) {
        if (value <= 0.04045f)
            return value * 1.f / 12.92f;
        return std::pow((value + 0.055f) * 1.f / 1.055f, (Float)2.4f);
    }
```

Refer to the “Further Reading” section for a more detailed discussion of gamma correction.

`InverseGammaCorrect()` is only applied when indicated via the `gamma` argument of `convertIn()`. By default, this is the case when the input image has an 8-bit color depth.

(ImageTexture Private Methods) ≡ 619

```
static void convertIn(const RGBSpectrum &from, RGBSpectrum *to,
                      Float scale, bool gamma) {
    for (int i = 0; i < RGBSpectrum::nSamples; ++i)
        (*to)[i] = scale * (gamma ? InverseGammaCorrect(from[i])
                               : from[i]);
}
static void convertIn(const RGBSpectrum &from, Float *to,
                      Float scale, bool gamma) {
    *to = scale * (gamma ? InverseGammaCorrect(from.y())
                           : from.y());
}
```

If the texture file wasn’t found or was unreadable, an image map with a single sample with a value of one is created so that the renderer can continue to generate an image of the scene without needing to abort execution. The `ReadImage()` function will issue a warning message in this case.

(Create one-valued MIPMap) ≡ 620

```
Tmemory oneVal = scale;
mipmap = new MIPMap<Tmemory>(Point2i(1, 1), &oneVal);
```

After the image is rendered and the system is cleaning up, the `ClearCache()` method is called to free the memory for the entries in the texture cache.

(ImageTexture Public Methods) ≡ 619

```
static void ClearCache() {
    textures.erase(textures.begin(), textures.end());
}
```

10.4.2 ImageTexture EVALUATION

The `ImageTexture::Evaluate()` routine does the usual texture coordinate computation and then hands the image map lookup to the `MIPMap`, which does the image filtering work for antialiasing. The returned value is still of type `Tmemory`; another conversion step similar to `ImageTexture::convertIn()` above converts to the returned type `Treturn`.

`Float` [1062](#)
`ImageTexture::convertIn()` [622](#)
`ImageTexture::Evaluate()` [623](#)
`InverseGammaCorrect()` [622](#)
`MIPMap` [625](#)
`Point2i` [68](#)
`ReadImage()` [1067](#)
`RGBSpectrum` [332](#)
`Spectrum::y()` [325](#)

```
<ImageTexture Public Methods> +≡ 619
    Treturn Evaluate(const SurfaceInteraction &si) const {
        Vector2f dstdx, dstdy;
        Point2f st = mapping->Map(si, &dstdx, &dstdy);
        Tmemory mem = mipmap->Lookup(st, dstdx, dstdy);
        Treturn ret;
        convertOut(mem, &ret);
        return ret;
    }
```

```
<ImageTexture Private Methods> +≡ 619
    static void convertOut(const RGBSpectrum &from, Spectrum *to) {
        Float rgb[3];
        from.ToRGB(rgb);
        *to = Spectrum::FromRGB(rgb);
    }
    static void convertOut(Float from, Float *to) {
        *to = from;
    }
```

10.4.3 MIP MAPS

As always, if the image function has higher frequency detail than can be represented by the texture sampling rate, aliasing will be present in the final image. Any frequencies higher than the Nyquist limit must be removed by prefiltering before the function is evaluated. Figure 10.9 shows the basic problem we face: an image texture has texels that are samples of some image function at a fixed frequency. The filter region for the lookup is given by its (s, t) center point and offsets to the estimated texture coordinate locations for the adjacent image samples. Because these offsets are estimates of the texture sampling

```
Float 1062
ImageTexture::mapping 619
ImageTexture::mipmap 619
MIPMap::Lookup() 635
Point2f 68
RGBSpectrum 332
Spectrum 315
Spectrum::FromRGB() 330
Spectrum::ToRGB() 328
SurfaceInteraction 116
TextureMapping2D::Map() 610
Vector2f 60
```

Figure 10.9: Given a point at which to perform an image map lookup (denoted by the solid point in the center) and estimates of the texture space sampling rate (denoted by adjacent solid points), it may be necessary to filter the contributions of a large number of texels in the image map (denoted by open points).

rate, we must remove any frequencies higher than twice the distance to the adjacent samples in order to satisfy the Nyquist criterion.

The texture sampling and reconstruction process has a few key differences from the image sampling process discussed in Chapter 7. These differences make it possible to address the antialiasing problem with more effective and less computationally expensive techniques. For example, here it is inexpensive to get the value of a sample—only an array lookup is necessary (as opposed to having to trace a number of rays to compute radiance). Further, because the texture image function is fully defined by the set of samples and there is no mystery about what its highest frequency could be, there is no uncertainty related to the function’s behavior between samples. These differences make it possible to remove detail from the texture before sampling, thus eliminating aliasing.

However, the texture sampling rate will typically change from pixel to pixel. The sampling rate is determined by scene geometry and its orientation, the texture coordinate mapping function, and the camera projection and image sampling rate. Because the texture sampling rate is not fixed, texture filtering algorithms need to be able to filter over arbitrary regions of texture samples efficiently.

The `MIPMap` class implements two methods for efficient texture filtering with spatially varying filter widths. The first, trilinear interpolation, is fast and easy to implement and was widely used for texture filtering in early graphics hardware. The second, elliptically weighted averaging, is slower and more complex but returns extremely high-quality results. Figure 10.1 shows the aliasing errors that result from ignoring texture filtering and just bilinearly interpolating texels from the most detailed level of the image map. Figure 10.10 shows the improvement from using the triangle filter and the EWA algorithm instead.

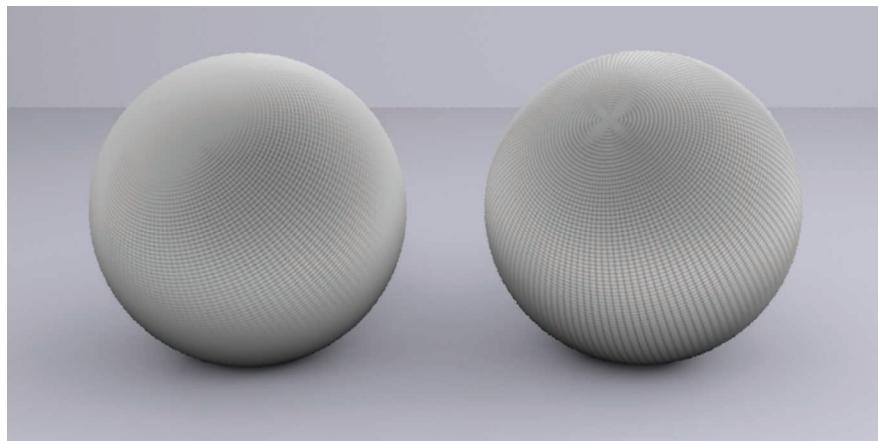


Figure 10.10: Filtering the image map properly substantially improves the image. On the left, trilinear interpolation was used; on the right, the EWA algorithm. Both of these approaches give a much better image than the unfiltered image map in Figure 10.1. Trilinear interpolation is inferior at handling strongly anisotropic filter footprints than EWA, which is why the edges of the sphere on the left are a uniform gray color (the overall average value of the texture), while details from the image map are visible farther along the edges of the sphere on the right before they fade to gray.

`MIPMap` 625

To limit the potential number of texels that need to be accessed, both of these filtering methods use an *image pyramid* of increasingly lower resolution prefiltered versions of the original image to accelerate their operation.³ The original image texels are at the bottom level of the pyramid, and the image at each level is half the resolution of the previous level, up to the top level, which has a single texel representing the average of all of the texels in the original image. This collection of images needs at most 1/3 more memory than storing the most detailed level alone and can be used to quickly find filtered values over large regions of the original image. The basic idea behind the pyramid is that if a large area of texels needs to be filtered a reasonable approximation is to use a higher level of the pyramid and do the filtering over the same area there, accessing many fewer texels.

```
<MIPMap Declarations> ≡
    template <typename T> class MIPMap {
        public:
            <MIPMap Public Methods 630>
        private:
            <MIPMap Private Methods 628>
            <MIPMap Private Data 625>
    };
```

In the constructor, the MIPMap copies the image data provided by the caller, resizes the image if necessary to ensure that its resolution is a power of two in each direction, and initializes a lookup table used by the elliptically weighted average filtering method in Section 10.4.5. It also records the desired behavior for texture coordinates that fall outside of the legal range in the wrapMode argument.

```
<MIPMap Method Definitions> ≡
    template <typename T>
    MIPMap<T>::MIPMap(const Point2i &res, const T *img, bool doTrilinear,
                        Float maxAnisotropy, ImageWrap wrapMode)
        : doTrilinear(doTrilinear), maxAnisotropy(maxAnisotropy),
          wrapMode(wrapMode), resolution(res) {
            std::unique_ptr<T[]> resampledImage = nullptr;
            if (!IsPowerOf2(resolution[0]) || !IsPowerOf2(resolution[1])) {
                (Resample image to power-of-two resolution 627)
            }
            <Initialize levels of MIPMap from image 630>
            <Initialize EWA filter weights if needed 639>
        }
```

```
<MIPMap Private Data> ≡
    const bool doTrilinear;
    const Float maxAnisotropy;
    const ImageWrap wrapMode;
    Point2i resolution;
```

Float 1062
ImageWrap 626
IsPowerOf2() 1064
MIPMap 625
Point2i 68

³ The name “MIP map” comes from the Latin *multum in parvo*, which means “much in little,” a nod to the image pyramid.

MIPMap is a template class that is parameterized by the data type of the image texels. pbrt creates MIPMaps of both `RGBSpectrum` and `Float` images; `Float` MIP maps are used for representing directional distributions of intensity from goniometric light sources (Section 12.3.3), for example. The `MIPMap` implementation requires that the type `T` support just a few basic operations, including addition and multiplication by a scalar.

The `ImageWrap` enumerant, passed to the `MIPMap` constructor, specifies the desired behavior when the supplied texture coordinates are not in the legal $[0, 1]$ range.

```
(MIPMap Helper Declarations) ≡
enum class ImageWrap { Repeat, Black, Clamp };
```

Implementation of an image pyramid is somewhat easier if the resolution of the original image is an exact power of two in each direction; this ensures that there is a straightforward relationship between the level of the pyramid and the number of texels at that level. If the user has provided an image where the resolution in one or both of the dimensions is not a power of two, then the `MIPMap` constructor starts by resizing the image up to the next power-of-two resolution greater than the original resolution before constructing the pyramid. Exercise 10.5 at the end of the chapter describes an approach to building image pyramids with non-power-of-two resolutions.

Image resizing here involves more application of the sampling and reconstruction theory from Chapter 7: we have an image function that has been sampled at one sampling rate, and we'd like to reconstruct a continuous image function from the original samples to resample at a new set of sample positions. Because this represents an increase in the sampling rate from the original rate, we don't have to worry about introducing aliasing due to undersampling high-frequency components in this step; we only need to reconstruct and directly resample the new function. Figure 10.11 illustrates this task in 1D.

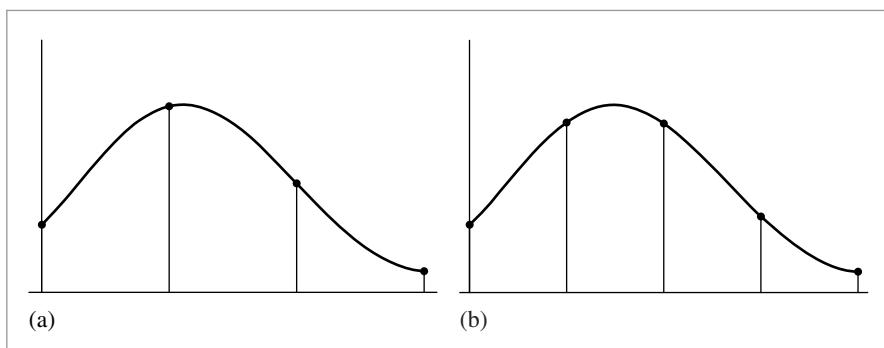


Figure 10.11: To increase an image's resolution to be a power of two, the `MIPMap` performs two 1D resampling steps with a separable reconstruction filter. (a) A 1D function reconstructed from four samples, denoted by dots. (b) To represent the same image function with more samples, we just need to reconstruct the continuous function and evaluate it at the new positions.

`ImageWrap` 626

`MIPMap` 625

`RGBSpectrum` 332

The MIPMap uses a separable reconstruction filter for this task; recall from Section 7.8 that separable filters can be written as the product of 1D filters: $f(x, y) = f(x)f(y)$. One advantage of using a separable filter is that if we are using one to resample an image from one resolution (s, t) to another (s', t') , then we can implement the resampling as two 1D resampling steps, first resampling in s to create an image of resolution (s', t) and then resampling that image to create the final image of resolution (s', t') . Resampling the image via two 1D steps in this manner simplifies implementation and makes the number of texels accessed for each texel in the final image a linear function of the filter width, rather than a quadratic one.

{Resample image to power-of-two resolution} ≡ 625
 Point2i resPow2(RoundUpPow2(resolution[0]), RoundUpPow2(resolution[1]));
{Resample image in s direction} 627
{Resample image in t direction}
 resolution = resPow2;

Reconstructing the original image function and sampling it at a new texel's position are mathematically equivalent to centering the reconstruction filter kernel at the new texel's position and weighting the nearby texels in the original image appropriately. Thus, each new texel is a weighted average of a small number of texels in the original image.

The MIPMap::resampleWeights() method determines which original texels contribute to each new texel and what the values are of the contribution weights for each new texel. It returns the values in an array of ResampleWeight structures for all of the texels in a 1D row or column of the image. Because this information is the same for all rows of the image when resampling in s and all columns when resampling in t , it's more efficient to compute it once for each of the two passes and then reuse it many times for each one. Given these weights, the image is first magnified in the s direction, turning the original image with resolution resolution into an image with resolution (resPow2[0], resolution[1]), which is stored in resampledImage. The implementation here allocates enough space in resampledImage to hold the final zoomed image with resolution (resPow2[0], resPow2[1]), so two large allocations can be avoided.

{Resample image in s direction} ≡ 627
 std::unique_ptr<ResampleWeight[]> sWeights =
 resampleWeights(resolution[0], resPow2[0]);
 resampledImage.reset(new T[resPow2[0] * resPow2[1]]);
{Apply sWeights to zoom in s direction} 629

```
Float 1062
MIPMap 625
MIPMap::resampleWeights()
  628
MIPMap::resolution 625
Point2i 68
ResampleWeight 627
RoundUpPow2() 1064
```

For the reconstruction filter used here, no more than four of the original texels will contribute to each new texel after zooming, so ResampleWeight only needs to hold four weights. Because the four texels are contiguous, we only store the offset to the first one.

{MIPMap Helper Declarations} +≡
 struct ResampleWeight {
 int firstTexel;
 Float weight[4];
};

```
(MIPMap Private Methods) ≡ 625
    std::unique_ptr<ResampleWeight[]> resampleWeights(int oldRes,
        int newRes) {
        Assert(newRes >= oldRes);
        std::unique_ptr<ResampleWeight[]> wt(new ResampleWeight[newRes]);
        Float filterwidth = 2.f;
        for (int i = 0; i < newRes; ++i) {
            {Compute image resampling weights for ith texel 628}
        }
        return wt;
    }
```

Just as it was important to distinguish between discrete and continuous pixel coordinates in Chapter 7, the same issues need to be addressed with texel coordinates here. We will use the same conventions as described in Section 7.1.7. For each new texel, this function starts by computing its continuous coordinates in terms of the old texel coordinates. This value is stored in center, because it is the center of the reconstruction filter for the new texel. Next, it is necessary to find the offset to the first texel that contributes to the new texel. This is a slightly tricky calculation—after subtracting the filter width to find the start of the filter’s nonzero range, it is necessary to add an extra 0.5 offset to the continuous coordinate before taking the floor to find the discrete coordinate. Figure 10.12 illustrates why this offset is needed.

Starting from this first contributing texel, this function loops over four texels, computing each one’s offset to the center of the filter kernel and the corresponding filter weight.

```
{Compute image resampling weights for ith texel} ≡ 628
    Float center = (i + .5f) * oldRes / newRes;
    wt[i].firstTexel = std::floor((center - filterwidth) + 0.5f);
    for (int j = 0; j < 4; ++j) {
        Float pos = wt[i].firstTexel + j + .5f;
        wt[i].weight[j] = Lanczos((pos - center) / filterwidth);
    }
    {Normalize filter weights for texel resampling 629}
```

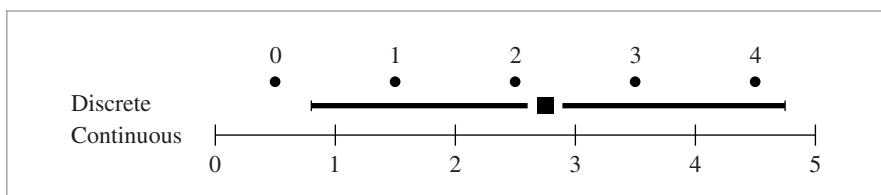


Figure 10.12: The computation to find the first texel inside a reconstruction filter’s support is slightly tricky. Consider a filter centered around continuous coordinate 2.75 with width 2, as shown here. The filter’s support covers the range [0.75, 4.75], although texel zero is outside the filter’s support: adding 0.5 to the lower end before taking the floor to find the discrete texel gives the correct starting texel, number one.

Assert() 1069
Float 1062
Lanczos() 629
ResampleWeight 627
ResampleWeight::firstTexel 627
ResampleWeight::weight 627

The reconstruction filter function used to compute the weights, `Lanczos()`, is the same as the one in `LanczosSincFilter::Sinc()`.

```
(Texture Declarations) +≡
    Float Lanczos(Float, Float tau = 2);
```

Depending on the filter function used, the four filter weights may not sum to one. Therefore, to ensure that the resampled image won't be any brighter or darker than the original image, the weights are normalized here.

```
(Normalize filter weights for texel resampling) ≡
    Float invSumWts = 1 / (wt[i].weight[0] + wt[i].weight[1] +
                           wt[i].weight[2] + wt[i].weight[3]);
    for (int j = 0; j < 4; ++j)
        wt[i].weight[j] *= invSumWts;
```

628

Once the weights have been computed, it's easy to apply them to compute the zoomed texels. For each of the `resolution[1]` horizontal scan lines in the original image, a pass is made across the `resPow2[0]` texels in the s -zoomed image using the precomputed weights to compute their values. Because the computation for each texel is completely independent of the computation for every other texel, and because this computation requires a bit of processing, it's worthwhile to split the image into sections and work on them in parallel with multiple threads.

```
(Apply sWeights to zoom in s direction) ≡
    ParallelFor(
        [&](int t) {
            for (int s = 0; s < resPow2[0]; ++s) {
                (Compute texel (s, t) in s-zoomed image 629)
            }
        }, resolution[1], 16);
```

627

The `ImageWrap` parameter to the `MIPMap` constructor determines the convention to be used for out-of-bounds texel coordinates. It either remaps them to valid values with a modulus or clamp calculation or uses a black texel value.

```
(Compute texel (s, t) in s-zoomed image) ≡
    resampledImage[t * resPow2[0] + s] = 0.f;
    for (int j = 0; j < 4; ++j) {
        int origS = sWeights[s].firstTexel + j;
        if (wrapMode == ImageWrap::Repeat)
            origS = Mod(origS, resolution[0]);
        else if (wrapMode == ImageWrap::Clamp)
            origS = Clamp(origS, 0, resolution[0] - 1);
        if (origS >= 0 && origS < (int)resolution[0])
            resampledImage[t * resPow2[0] + s] +=
                sWeights[s].weight[j] * img[t * resolution[0] + origS];
    }
```

629

`Clamp()` 1062
`Float` 1062
`ImageWrap` 626
`Lanczos()` 629
`LanczosSincFilter::Sinc()` 483
`MIPMap` 625
`Mod()` 1062
`ParallelFor()` 1088
`ResampleWeight::firstTexel` 627
`ResampleWeight::weight` 627

The process for resampling in the t direction is almost the same as for s , so we won't include the implementation here.

Once we have an image with resolutions that are powers of two, the levels of the MIP map can be initialized, starting from the bottom (finest) level. Each higher level is found by filtering the texels from the previous level.

Because image maps use a fair amount of memory, and because 8 to 20 texels are typically used per image texture lookup to compute a filtered value, it's worth carefully considering how the texels are laid out in memory, since reducing cache misses while accessing the texture map can noticeably improve the renderer's performance. Because both of the texture filtering methods implemented in this section access a set of texels in a rectangular region of the image map each time a lookup is performed, the `MIPMap` uses the `BlockedArray` template class to store the 2D arrays of texel values, rather than a standard C++ array. The `BlockedArray` reorders the array values in memory in a way that improves cache coherence when the values are accessed with these kinds of rectangular patterns; it is described in Section A.4.4 in Appendix A.

(Initialize levels of MIPMap from image) ≡ 625

```
int nLevels = 1 + Log2Int(std::max(resolution[0], resolution[1]));
pyramid.resize(nLevels);
(Initialize most detailed level of MIPMap 630)
for (int i = 1; i < nLevels; ++i) {
    (Initialize ith MIPMap level from i - 1st level 631)
}
```

(MIPMap Private Data) +≡ 625

```
std::vector<std::unique_ptr<BlockedArray<T>>> pyramid;
```

(MIPMap Public Methods) ≡ 625

```
int Width() const { return resolution[0]; }
int Height() const { return resolution[1]; }
int Levels() const { return pyramid.size(); }
```

The base level of the MIP map, which holds the original data (or the resampled data, if it didn't originally have power-of-two resolutions), is initialized by the default `BlockedArray` constructor.

(Initialize most detailed level of MIPMap) ≡ 630

```
pyramid[0].reset(new BlockedArray<T>(resolution[0], resolution[1],
    resampledImage ? resampledImage.get() : img));
```

Before showing how the rest of the levels are initialized, we will first define a texel access function that will be used during that process. `MIPMap::Texel()` returns a reference to the texel value for the given discrete integer-valued texel position. As described earlier, if an out-of-range texel coordinate is passed in, then based on the value of `wrapMode`, this method effectively repeats the texture over the entire 2D texture coordinate domain by taking the modulus of the coordinate with respect to the texture size, clamps the

`BlockedArray` 1076
`Log2Int()` 1064
`MIPMap` 625
`MIPMap::pyramid` 630
`MIPMap::resolution` 625
`MIPMap::Texel()` 631

coordinates to the valid range so that the border pixels are used, or returns a black texel for out-of-bounds coordinates.

```
(MIPMap Method Definitions) +≡
template <typename T>
const T &MIPMap<T>::Texel(int level, int s, int t) const {
    const BlockedArray<T> &l = *pyramid[level];
    (Compute texel (s, t) accounting for boundary conditions 631)
    return l(s, t);
}
```

(Compute texel (s, t) accounting for boundary conditions) ≡

631

```
switch (wrapMode) {
    case ImageWrap::Repeat:
        s = Mod(s, l.uSize());
        t = Mod(t, l.vSize());
        break;
    case ImageWrap::Clamp:
        s = Clamp(s, 0, l.uSize() - 1);
        t = Clamp(t, 0, l.vSize() - 1);
        break;
    case ImageWrap::Black:
        static const T black = 0.f;
        if (s < 0 || s >= (int)l.uSize() ||
            t < 0 || t >= (int)l.vSize())
            return black;
        break;
}
```

For non-square images, the resolution in one direction must be clamped to one for the upper levels of the image pyramid, where there is still downsampling to do in the larger of the two resolutions. This is handled by the following `std::max()` calls:

(Initialize i th MIPMap level from i - 1st level) ≡

630

```
BlockedArray 1076
BlockedArray::uSize() 1078
BlockedArray::vSize() 1078
Clamp() 1062
ImageWrap 626
MIPMap 625
MIPMap::pyramid 630
Mod() 1062
```

The MIPMap uses a simple box filter to average four texels from the previous level to find the value at the current texel. Using the Lanczos filter here would give a slightly better result for this computation, although this modification is left for Exercise 10.4 at the end of the chapter. As with resampling to power-of-two resolutions, doing this downfiltering using multiple threads rather than with a regular `for` loop is worthwhile here.

(Filter four texels from finer level of pyramid) ≡

631

```
ParallelFor(
[&](int t) {
    for (int s = 0; s < sRes; ++s)
        (*pyramid[i])(s, t) = .25f *
            (Texel(i-1, 2*s, 2*t) + Texel(i-1, 2*s+1, 2*t) +
             Texel(i-1, 2*s, 2*t+1) + Texel(i-1, 2*s+1, 2*t+1));
}, tRes, 16);
```

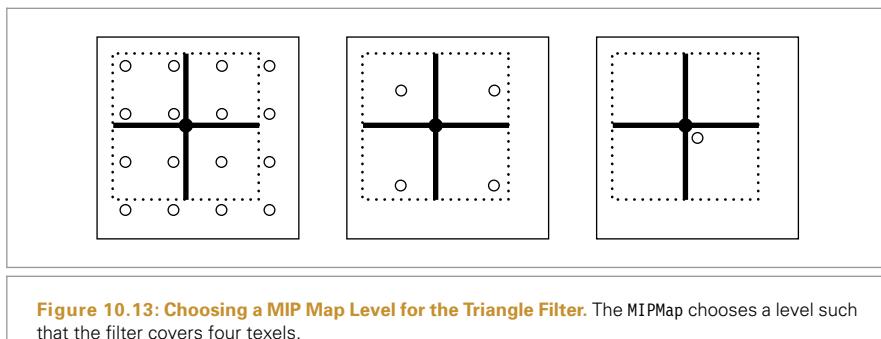
10.4.4 ISOTROPIC TRIANGLE FILTER

The first of the two `MIPMap::Lookup()` methods uses a triangle filter over the texture samples to remove high frequencies. Although this filter function does not give high-quality results, it can be implemented very efficiently. In addition to the (s, t) coordinates of the evaluation point, the caller passes this method a filter width for the lookup, giving the extent of the region of the texture to filter across. This method filters over a square region in texture space, so the width should be conservatively chosen to avoid aliasing in both the s and t directions. Filtering techniques like this one that do not support a filter extent that is non-square or non-axis-aligned are known as *isotropic*. The primary disadvantage of isotropic filtering algorithms is that textures viewed at an oblique angle will appear blurry, since the required sampling rate along one axis will be very different from the sampling rate along the other in this case.

Because filtering over many texels for wide filter widths would be inefficient, this method chooses a MIP map level from the pyramid such that the filter region at that level would cover four texels at that level. Figure 10.13 illustrates this idea.

(MIPMap Method Definitions) +≡

```
template <typename T>
T MIPMap<T>::Lookup(const Point2f &st, Float width) const {
    (Compute MIPMap level for trilinear filtering 633)
    (Perform trilinear interpolation at appropriate MIPMap level 633)
}
```



Float 1062
MIPMap 625
MIPMap::Lookup() 633
MIPMap::pyramid 630
MIPMap::Texel() 631
ParallelFor() 1088
Point2f 68

Since the resolutions of the levels of the pyramid are all powers of two, the resolution of level l is $2^{\text{nLevels}-1-l}$. Therefore, to find the level with a texel spacing width w requires solving

$$\frac{1}{w} = 2^{\text{nLevels}-1-l}$$

for l . In general, this will be a floating-point value between two MIP map levels.

```
<Compute MIPMap level for trilinear filtering> ≡
    Float level = Levels() - 1 + Log2(std::max(width, (Float)1e-8));
```

632

As shown by Figure 10.13, applying a triangle filter to the four texels around the sample point will either filter over too small a region or too large a region (except for very carefully selected filter widths). The implementation here applies the triangle filter at both of these levels and blends between them according to how close `level` is to each of them. This helps hide the transitions from one MIP map level to the next at nearby pixels in the final image. While applying a triangle filter to four texels at two levels in this manner doesn't give exactly the same result as applying it to the original highest resolution texels, the difference isn't too bad in practice, and the efficiency of this approach is worth this penalty. In any case, the elliptically weighted average filtering in the next section should be used when texture quality is important.

```
<Perform trilinear interpolation at appropriate MIPMap level> ≡
    if (level < 0)
        return triangle(0, st);
    else if (level >= Levels() - 1)
        return Texel(Levels() - 1, 0, 0);
    else {
        int iLevel = std::floor(level);
        Float delta = level - iLevel;
        return Lerp(delta, triangle(iLevel, st), triangle(iLevel + 1, st));
    }
```

632

Given floating-point texture coordinates in $[0, 1]^2$, the `MIPMap::triangle()` routine uses a triangle filter to interpolate between the four texels that surround the sample point, as shown in Figure 10.14. This method first scales the coordinates by the texture resolution at the given MIP map level in each direction, turning them into continuous texel coordinates. Because these are continuous coordinates, but the texels in the image map are defined at discrete texture coordinates, it's important to carefully convert into a common representation. Here, we will do all of our work in discrete coordinates, mapping the continuous texture coordinates to discrete space.

For example, consider the 1D case with a continuous texture coordinate of 2.4: this coordinate is a distance of 0.1 below the discrete texel coordinate 2 (which corresponds to a continuous coordinate of 2.5) and is 0.9 above the discrete coordinate 1 (continuous coordinate 1.5). Thus, if we subtract 0.5 from the continuous coordinate 2.4, giving 1.9, we can correctly compute the correct distances to the discrete coordinates 1 and 2 by subtracting coordinates.

`Float` 1062
`Log2()` 1063
`MIPMap::Levels()` 630
`MIPMap::Texel()` 631
`MIPMap::triangle()` 634
`Spectrum::Lerp()` 317

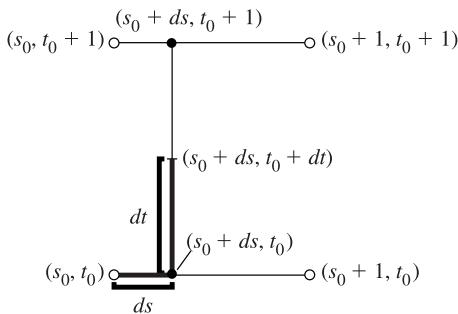


Figure 10.14: To compute the value of the image texture function at an arbitrary (s, t) position, `MIPMap::triangle()` finds the four texels around (s, t) and weights them according to a triangle filter based on their distance to (s, t) . One way to implement this is as a series of linear interpolations, as shown here: First, the two texels below (s, t) are linearly interpolated to find a value at $(s, 0)$, and the two texels above it are interpolated to find $(s, 1)$. Then, $(s, 0)$ and $(s, 1)$ are linearly interpolated again to find the value at (s, t) .

After computing the distances in s and t to the texel at the lower left of the given coordinates, ds and dt , `MIPMap::triangle()` determines weights for the four texels and computes the filtered value. Recall that the triangle filter is

$$f(x, y) = (1 - |x|)(1 - |y|);$$

the appropriate weights follow directly. Notice the similarity between this computation and `BilerpTexture::Evaluate()`.

(MIPMap Method Definitions) +≡

```
template <typename T>
T MIPMap<T>::triangle(int level, const Point2f &st) const {
    level = Clamp(level, 0, Levels() - 1);
    Float s = st[0] * pyramid[level]->uSize() - 0.5f;
    Float t = st[1] * pyramid[level]->vSize() - 0.5f;
    int s0 = std::floor(s), t0 = std::floor(t);
    Float ds = s - s0, dt = t - t0;
    return (1 - ds) * (1 - dt) * Texel(level, s0, t0) +
           (1 - ds) * dt      * Texel(level, s0, t0+1) +
           ds      * (1 - dt) * Texel(level, s0+1, t0) +
           ds      * dt      * Texel(level, s0+1, t0+1);
}
```

BilerpTexture::Evaluate()
618
BlockedArray::uSize() 1078
BlockedArray::vSize() 1078
Clamp() 1062
Float 1062
MIPMap 625
MIPMap::Levels() 630
MIPMap::pyramid 630
MIPMap::Texel() 631
MIPMap::triangle() 634
Point2f 68

* 10.4.5 ELLIPTICALLY WEIGHTED AVERAGE

The elliptically weighted average (EWA) algorithm fits an ellipse to the two axes in texture space given by the texture coordinate differentials and then filters the texture with a Gaussian filter function (Figure 10.15). It is widely regarded as one of the best texture filtering algorithms in graphics and has been carefully derived from the basic principles of sampling theory. Unlike the triangle filter in the previous section, it can filter over

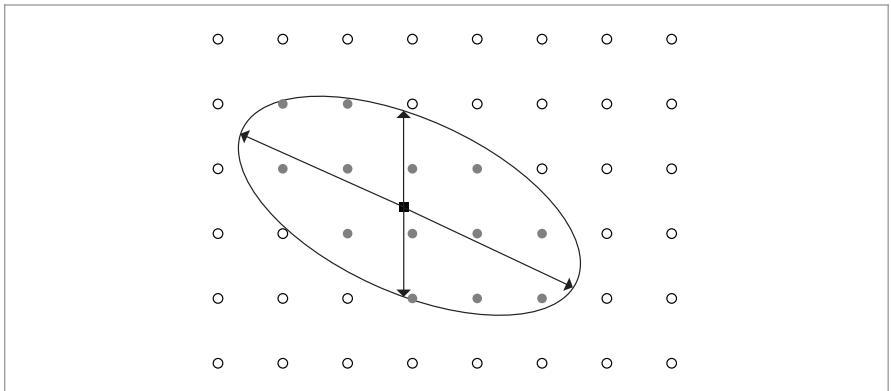


Figure 10.15: The EWA filter applies a Gaussian filter to the texels in an elliptical area around the evaluation point. The extent of the ellipse is such that its edge passes through the positions of the adjacent texture samples as estimated by the texture coordinate partial derivatives.

arbitrarily oriented regions of the texture, with some flexibility of having different filter extents in different directions. This type of filter is known as *anisotropic*. This capability greatly improves the quality of its results, since it can properly adapt to different sampling rates along the two image axes.

We won't show the full derivation of this filter here, although we do note that it is distinguished by being a *unified resampling filter*: it simultaneously computes the result of a Gaussian filtered texture function convolved with a Gaussian reconstruction filter in image space. This is in contrast to many other texture filtering methods that ignore the effect of the image space filter or equivalently assume that it is a box. Even if a Gaussian isn't being used for filtering the samples for the image being rendered, taking some account of the spatial variation of the image filter improves the results, assuming that the filter being used is somewhat similar in shape to the Gaussian, as the Mitchell and windowed sinc filters are.

```

⟨MIPMap Method Definitions⟩ +≡
    template <typename T>
    T MIPMap<T>::Lookup(const Point2f &st, Vector2f dst0,
                           Vector2f dst1) const {
        if (doTrilinear) {
            Float width = std::max(std::max(std::abs(dst0[0]),
                                             std::abs(dst0[1])),
                                   std::max(std::abs(dst1[0]),
                                             std::abs(dst1[1])));
            return Lookup(st, 2 * width);
        }
        ⟨Compute ellipse minor and major axes 636⟩
        ⟨Clamp ellipse eccentricity if too large 636⟩
        ⟨Choose level of detail for EWA lookup and perform EWA filtering 636⟩
    }

```

The screen space partial derivatives of the texture coordinates define the axes of the ellipse. The lookup method starts out by determining which of the two axes is the major axis (the longer of the two) and which is the minor, swapping them if needed so that `dst0` is the major axis. The length of the minor axis will be used shortly to select a MIP map level.

```
(Compute ellipse minor and major axes) ≡
if (dst0.LengthSquared() < dst1.LengthSquared())
    std::swap(dst0, dst1);
Float majorLength = dst0.Length();
Float minorLength = dst1.Length();
```

635

Next the *eccentricity* of the ellipse is computed—the ratio of the length of the major axis to the length of the minor axis. A large eccentricity indicates a very long and skinny ellipse. Because this method filters texels from a MIP map level chosen based on the length of the minor axis, highly eccentric ellipses mean that a large number of texels need to be filtered. To avoid this expense (and to ensure that any EWA lookup takes a bounded amount of time), the length of the minor axis may be increased to limit the eccentricity. The result may be an increase in blurring, although this effect usually isn't noticeable in practice.

```
(Clamp ellipse eccentricity if too large) ≡
if (minorLength * maxAnisotropy < majorLength && minorLength > 0) {
    Float scale = majorLength / (minorLength * maxAnisotropy);
    dst1 *= scale;
    minorLength *= scale;
}
if (minorLength == 0)
    return triangle(0, st);
```

635

Like the triangle filter, the EWA filter uses the image pyramid to reduce the number of texels to be filtered for a particular texture lookup, choosing a MIP map level based on the length of the minor axis. Given the limited eccentricity of the ellipse due to the clamping above, the total number of texels used is thus bounded. Given the length of the minor axis, the computation to find the appropriate pyramid level is the same as was used for the triangle filter. Similarly, the implementation here blends between the filtered results at the two levels around the computed level of detail, again to reduce artifacts from transitions from one level to another.

```
(Choose level of detail for EWA lookup and perform EWA filtering) ≡
Float lod = std::max((Float)0, Levels() - (Float)1 + Log2(minorLength));
int ilod = std::floor(lod);
return Lerp(lod - ilod, EWA(ilod, st, dst0, dst1),
            EWA(ilod + 1, st, dst0, dst1));
```

Float 1062 Log2() 1063 MIPMap::EWA() 637 MIPMap::Levels() 630 MIPMap::maxAnisotropy 625 MIPMap::triangle() 634 Spectrum::Lerp() 317

The `MIPMap::EWA()` method actually applies the filter at a particular level.

```

⟨MIPMap Method Definitions⟩ +≡
    template <typename T>
    T MIPMap<T>::EWA(int level, Point2f st, Vector2f dst0,
                        Vector2f dst1) const {
        if (level >= Levels()) return Texel(Levels() - 1, 0, 0);
        ⟨Convert EWA coordinates to appropriate scale for level 637⟩
        ⟨Compute ellipse coefficients to bound EWA filter region 637⟩
        ⟨Compute the ellipse's (s, t) bounding box in texture space 638⟩
        ⟨Scan over ellipse bound and compute quadratic equation 639⟩
    }

```

This method first converts from texture coordinates in [0, 1] to coordinates and differentials in terms of the resolution of the chosen MIP map level. It also subtracts 0.5 from the continuous position coordinate to align the sample point with the discrete texel coordinates, as was done in `MIPMap::triangle()`.

```

⟨Convert EWA coordinates to appropriate scale for level⟩ ≡
    st[0] = st[0] * pyramid[level]->uSize() - 0.5f;
    st[1] = st[1] * pyramid[level]->vSize() - 0.5f;
    dst0[0] *= pyramid[level]->uSize();
    dst0[1] *= pyramid[level]->vSize();
    dst1[0] *= pyramid[level]->uSize();
    dst1[1] *= pyramid[level]->vSize();

```

637

It next computes the coefficients of the implicit equation for the ellipse with axes (ds_0, dt_0) and (ds_1, dt_1) and centered at the origin. Placing the ellipse at the origin rather than at (s, t) simplifies the implicit equation and the computation of its coefficients and can be easily corrected for when the equation is evaluated later. The general form of the implicit equation for all points (s, t) inside such an ellipse is

$$e(s, t) = As^2 + Bst + Ct^2 < F,$$

although it is more computationally efficient to divide through by F and express this as

$$e(s, t) = \frac{A}{F}s^2 + \frac{B}{F}st + \frac{C}{F}t^2 = A's^2 + B'st + C't^2 < 1.$$

We will not derive the equations that give the values of the coefficients, although the interested reader can easily verify their correctness.⁴

```

BlockedArray::uSize() 1078
BlockedArray::vSize() 1078
Float 1062
MIPMap 625
MIPMap::Levels() 630
MIPMap::pyramid 630
MIPMap::Texel() 631
MIPMap::triangle() 634
Point2f 68
Vector2f 60

```

637

```

⟨Compute ellipse coefficients to bound EWA filter region⟩ ≡
    Float A = dst0[1] * dst0[1] + dst1[1] * dst1[1] + 1;
    Float B = -2 * (dst0[0] * dst0[1] + dst1[0] * dst1[1]);
    Float C = dst0[0] * dst0[0] + dst1[0] * dst1[0] + 1;
    Float invF = 1 / (A * C - B * B * 0.25f);
    A *= invF;
    B *= invF;
    C *= invF;

```

⁴ Heckbert's thesis has the original derivation (Heckbert 1989, p. 80). A and C have an extra term of 1 added to them so the ellipse is a minimum of one texel separation wide. This ensures that the ellipse will not fall between the texels when magnifying at the most detailed level.

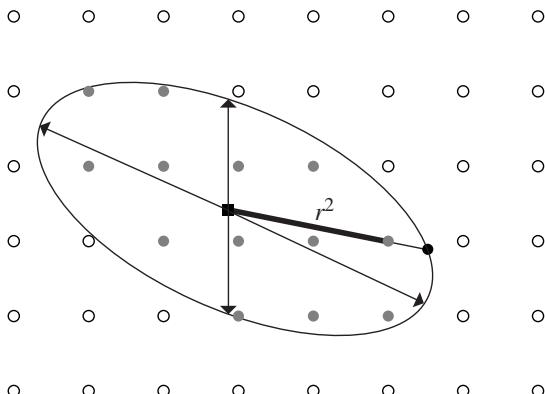


Figure 10.16: Finding the r^2 Ellipse Value for the EWA Filter Table Lookup.

The next step is to find the axis-aligned bounding box in discrete integer texel coordinates of the texels that are potentially inside the ellipse. The EWA algorithm loops over all of these candidate texels, filtering the contributions of those that are in fact inside the ellipse. The bounding box is found by determining the minimum and maximum values that the ellipse takes in the s and t directions. These extrema can be calculated by finding the partial derivatives $\partial e / \partial s$ and $\partial e / \partial t$, finding their solutions for $s = 0$ and $t = 0$, and adding the offset to the ellipse center. For brevity, we will not include the derivation for these expressions here.

(Compute the ellipse's (s, t) bounding box in texture space) \equiv

637

```
Float det = -B * B + 4 * A * C;
Float invDet = 1 / det;
Float uSqrt = std::sqrt(det * C), vSqrt = std::sqrt(A * det);
int s0 = std::ceil(st[0] - 2 * invDet * uSqrt);
int s1 = std::floor(st[0] + 2 * invDet * uSqrt);
int t0 = std::ceil(st[1] - 2 * invDet * vSqrt);
int t1 = std::floor(st[1] + 2 * invDet * vSqrt);
```

Now that the bounding box is known, the EWA algorithm loops over the texels, transforming each one to the coordinate system where the texture lookup point (s, t) is at the origin with a translation. It then evaluates the ellipse equation to see if the texel is inside the ellipse (Figure 10.16) and computes the filter weight for the texel if so. The final filtered value returned is a weighted sum over texels (s', t') inside the ellipse, where f is the Gaussian filter function:

Float 1062

$$\frac{\sum f(s' - s, t' - t)t(s', t')}{\sum f(s' - s, t' - t)}.$$

```

<Scan over ellipse bound and compute quadratic equation> ≡ 637
    T sum(0.f);
    Float sumWts = 0;
    for (int it = t0; it <= t1; ++it) {
        Float tt = it - st[1];
        for (int is = s0; is <= s1; ++is) {
            Float ss = is - st[0];
            (Compute squared radius and filter texel if inside ellipse 639)
        }
    }
    return sum / sumWts;

```

A nice feature of the implicit equation $e(s, t)$ is that its value at a particular texel is the squared ratio of the distance from the center of the ellipse to the texel to the distance from the center of the ellipse to the ellipse boundary along the line through that texel (Figure 10.16). This value can be used to index into a precomputed lookup table of Gaussian filter function values.

```

<Compute squared radius and filter texel if inside ellipse> ≡ 639
    Float r2 = A * ss * ss + B * ss * tt + C * tt * tt;
    if (r2 < 1) {
        int index = std::min((int)(r2 * WeightLUTSize),
                             WeightLUTSize - 1);
        Float weight = weightLut[index];
        sum += Texel(level, is, it) * weight;
        sumWts += weight;
    }

```

The lookup table is initialized the first time a MIPMap is constructed. Because it will be indexed with squared distances from the filter center r^2 , each entry stores a value $e^{-\alpha r^2}$, rather than $e^{-\alpha r^2}$.

```

<MIPMap Private Data> +≡ 625
    static constexpr int WeightLUTSize = 128;
    static Float weightLut[WeightLUTSize];

```

So that the filter function goes to zero at the end of its extent rather than having an abrupt step, `std::exp(-alpha)` is subtracted from the filter values here.

```

<Initialize EWA filter weights if needed> ≡ 625
    if (weightLut[0] == 0.) {
        for (int i = 0; i < WeightLUTSize; ++i) {
            Float alpha = 2;
            Float r2 = Float(i) / Float(WeightLUTSize - 1);
            weightLut[i] = std::exp(-alpha * r2) - std::exp(-alpha);
        }
    }

```

10.5 SOLID AND PROCEDURAL TEXTURING

Once one starts to think of the (s, t) texture coordinates used by 2D texture functions as quantities that can be computed by arbitrary functions and not just from the parametric coordinates of the surface, it is natural to generalize texture functions to be defined over 3D domains (often called *solid textures*) rather than just 2D (s, t) . One reason solid textures are particularly convenient is that all objects have a natural 3D texture mapping—object space position. This is a substantial advantage for texturing objects that don’t have a natural 2D parameterization (e.g., triangle meshes and implicit surfaces) and for objects that have a distorted parameterization (e.g., near the poles of a sphere). In preparation for this idea, Section 10.2.5 defined a general `TextureMapping3D` interface to compute 3D texture coordinates as well as a `TransformMapping3D` implementation.

Solid textures introduce a new problem, however: texture representation. A 3D image map takes up a fair amount of storage space and is much harder to acquire than a 2D texture map, which can be extracted from photographs or painted by an artist. Therefore, procedural texturing—the idea that programs could be executed to generate texture values at arbitrary positions on surfaces in the scene—came into use at the same time that solid texturing was developed. A simple example of procedural texturing is a procedural sine wave. If we wanted to use a sine wave for bump mapping (for example, to simulate waves in water), it would be inefficient and potentially inaccurate to precompute values of the function at a grid of points and then store them in an image map. Instead, it makes much more sense to evaluate the `sin()` function at points on the surface as needed.

If we can find a 3D function that describes the colors of the grain in a solid block of wood, for instance, then we can generate images of complex objects that appear to be carved from wood. Over the years, procedural texturing has grown in application considerably as techniques have been developed to describe more and more complex surfaces procedurally.

Procedural texturing has a number of interesting implications. First, it can be used to reduce memory requirements for rendering, by reducing the need for the storage of large, high-resolution texture maps. In addition, procedural shading gives the promise of potentially infinite detail; as the viewer approaches an object, the texturing function is evaluated at the points being shaded, which naturally leads to the right amount of detail being visible. In contrast, image texture maps become blurry when the viewer is too close to them. On the other hand, subtle details of the appearance of procedural textures can be much more difficult to control than when image maps are used.

Another challenge with procedural textures is antialiasing. Procedural textures are often expensive to evaluate, and sets of point samples that fully characterize their behavior aren’t available as they are for image maps. Because we would like to remove high-frequency information in the texture function before we take samples from it, we need to be aware of the frequency content of the various steps we take along the way so we can avoid introducing high frequencies. Although this sounds daunting, there are a handful of techniques that work well to handle this issue.

`TextureMapping3D` 614

`TransformMapping3D` 614

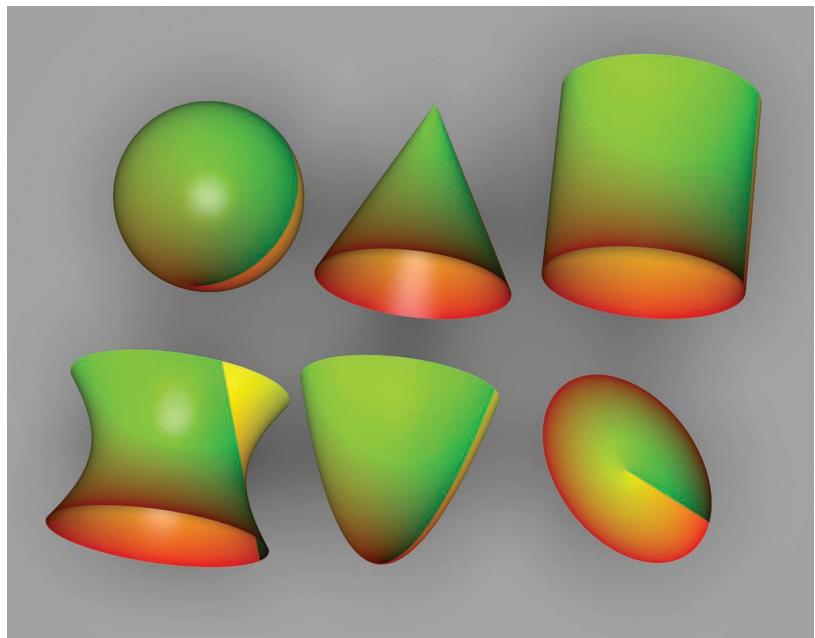


Figure 10.17: The UV Texture Applied to All of pbrt's Quadric Shapes. The u parameter is mapped to the red channel, and the v parameter is mapped to green.

10.5.1 UV TEXTURE

Our first procedural texture converts the surface's (u, v) coordinates into the red and green components of a `Spectrum` (Figure 10.17). It is especially useful when debugging the parameterization of a new `Shape`, for example. It is defined in `textures/uv.h` and `textures/uv.cpp`.

```

⟨UVTexture Declarations⟩ ≡
class UVTexture : public Texture<Spectrum> {
public:
    ⟨UVTexture Public Methods 641⟩
private:
    std::unique_ptr<TextureMapping2D> mapping;
};

⟨UVTexture Public Methods⟩ ≡
Spectrum Evaluate(const SurfaceInteraction &si) const {
    Vector2f dstdx, dstdy;
    Point2f st = mapping->Map(si, &dstdx, &dstdy);
    Float rgb[3] =
        { st[0] - std::floor(st[0]), st[1] - std::floor(st[1]), 0 };
    return Spectrum::FromRGB(rgb);
}

```

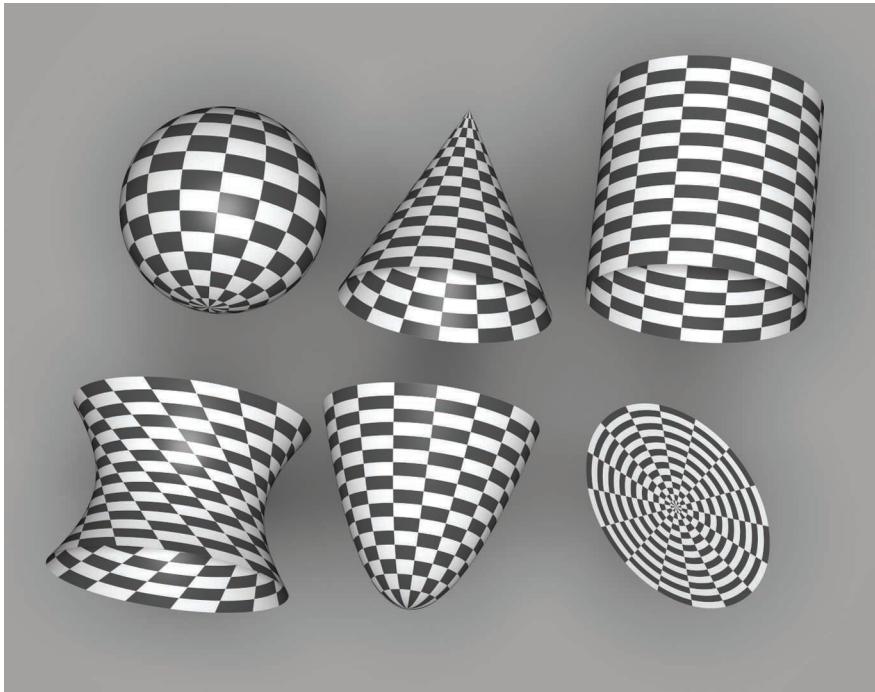


Figure 10.18: The Checkerboard Texture Applied to All of pbrt's Quadric Shapes.

10.5.2 CHECKERBOARD

The checkerboard is the canonical procedural texture (Figure 10.18). The (s, t) texture coordinates are used to break up parameter space into square regions that are shaded with alternating patterns. Rather than just supporting checkerboards that switch between two fixed colors, the implementation here allows the user to pass in two textures to color the alternating regions. The traditional black-and-white checkerboard is obtained by passing two `ConstantTexture`s. Its implementation is in the files `textures/checkerboard.h` and `textures/checkerboard.cpp`.

(CheckerboardTexture Declarations) ≡

```
template <typename T> class Checkerboard2DTexture : public Texture<T> {
public:
    (Checkerboard2DTexture Public Methods 643)
private:
    (Checkerboard2DTexture Private Data 643)
};
```

For simplicity, the frequency of the check function is 1 in (s, t) space: checks are one unit wide in each direction. The effective frequency can always be changed by the `TextureMapping2D` class with an appropriate scale of the (s, t) coordinates.

`ConstantTexture` 615

`Texture` 614

`TextureMapping2D` 609

```

⟨Checkerboard2DTexture Public Methods⟩ ≡ 642
    Checkerboard2DTexture(std::unique_ptr<TextureMapping2D> mapping,
        const std::shared_ptr<Texture<T>> &tex1,
        const std::shared_ptr<Texture<T>> &tex2, AAMethod aaMethod)
    : mapping(std::move(mapping)), tex1(tex1), tex2(tex2),
      aaMethod(aaMethod) { }

⟨Checkerboard2DTexture Private Data⟩ ≡ 642
    std::unique_ptr<TextureMapping2D> mapping;
    const std::shared_ptr<Texture<T>> tex1, tex2;
    const AAMethod aaMethod;

```

The checkerboard is good for demonstrating trade-offs between various antialiasing approaches for procedural textures. The implementation here supports both simple point sampling (no antialiasing) and a closed-form box filter evaluated over the filter region. The image sequence in Figure 10.23 at the end of this section shows the results of these approaches. The aaMethod enum constant selects which approach is used.

```

⟨AAMethod Declaration⟩ ≡
enum class AAMethod { None, ClosedForm };

```

The evaluation routine does the usual texture coordinate and differential computation and then uses the appropriate fragment to compute an antialiased checkerboard value (or not antialiased, if point sampling has been selected).

```

⟨Checkerboard2DTexture Public Methods⟩ +≡ 642
    T Evaluate(const SurfaceInteraction &si) const {
        Vector2f dstdx, dstdy;
        Point2f st = mapping->Map(si, &dstdx, &dstdy);
        if (aaMethod == AAMethod::None) {
            ⟨Point sample Checkerboard2DTexture 643⟩
        } else {
            ⟨Compute closed-form box-filtered Checkerboard2DTexture value 644⟩
        }
    }

AAMethod 643
AAMethod::None 643
Checkerboard2DTexture 642
Checkerboard2DTexture::aaMethod 643
Checkerboard2DTexture::mapping 643
Checkerboard2DTexture::tex1 643
Checkerboard2DTexture::tex2 643
Point2f 68
SurfaceInteraction 116
Texture 614
Texture::Evaluate() 615
TextureMapping2D 609
TextureMapping2D::Map() 610
Vector2f 60

```

The simplest case is to ignore antialiasing and just point-sample the checkerboard texture at the point. For this case, after getting the (s, t) texture coordinates from the TextureMapping2D, the integer checkerboard coordinates for that (s, t) position are computed, added together, and checked for odd or even parity to determine which of the two textures to evaluate.

```

⟨Point sample Checkerboard2DTexture⟩ ≡ 643, 645
    if (((int)std::floor(st[0]) + (int)std::floor(st[1])) % 2 == 0)
        return tex1->Evaluate(si);
    return tex2->Evaluate(si);

```

Given how bad aliasing can be in a point-sampled checkerboard texture, we will invest some effort to antialias it properly. The easiest case happens when the entire filter region lies inside a single check (Figure 10.19). In this case, we simply need to determine which of the check types we are inside and evaluate that one. As long as the Texture inside

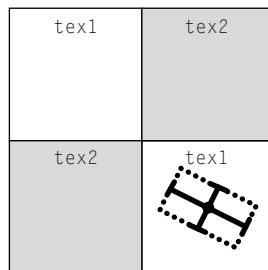


Figure 10.19: The Easy Case for Filtering the Checkerboard. If the filter region around the lookup point is entirely in one check, the checkerboard texture doesn't need to worry about antialiasing and can just evaluate the texture for that check.

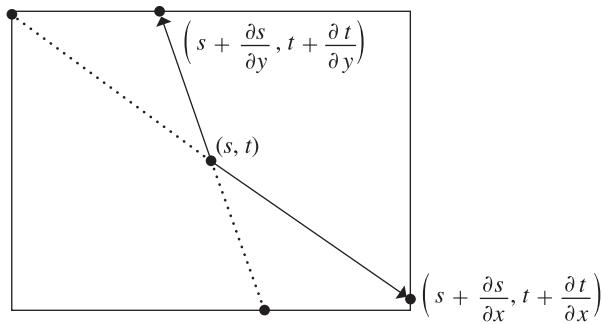


Figure 10.20: It is often convenient to use the axis-aligned bounding box around the texture evaluation point and the offsets from its partial derivatives as the region to filter over. Here, it's easy to see that the lengths of sides of the box are $2 \max(|\partial s/\partial x|, |\partial s/\partial y|)$ and $2 \max(|\partial t/\partial x|, |\partial t/\partial y|)$.

that check does appropriate antialiasing itself, the result for this case will be properly antialiased.

(Compute closed-form box-filtered Checkerboard2DTexture value) ≡

643

(Evaluate single check if filter is entirely inside one of them 645)

(Apply box filter to checkerboard region 646)

It's straightforward to check if the entire filter region is inside a single check by computing its bounding box and seeing if its extent lies inside the same check. For the remainder of this section, we will use the axis-aligned bounding box of the filter region given by the partial derivatives $\partial s/\partial x$, $\partial s/\partial y$, and so on, as the area to filter over, rather than trying to filter over the ellipse defined by the partial derivatives as the EWA filter did (Figure 10.20). This simplifies the implementation here, although somewhat increases the blurriness of

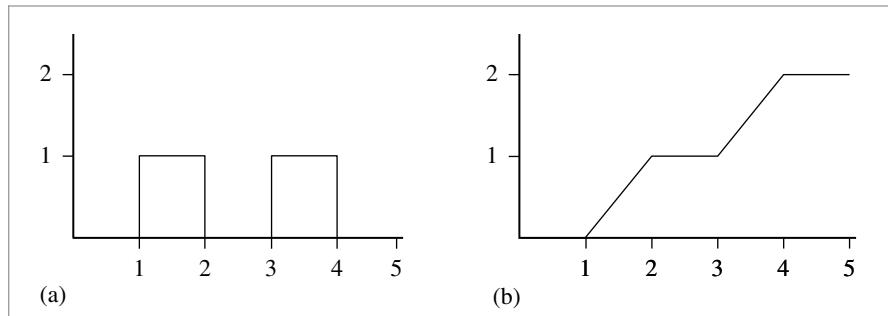


Figure 10.21: Integrating the Checkerboard Step Function. (a) The 1D step function that defines the checkerboard texture function, $c(x)$. (b) A graph of the value of the integral $\int_0^x c(x) dx$.

the filtered values. The variables ds and dt in the following hold half the filter width in each direction, so the total area filtered over ranges from $(s-ds, t-dt)$ to $(s+ds, t+dt)$.

{Evaluate single check if filter is entirely inside one of them} ≡

644

```
Float ds = std::max(std::abs(dstdx[0]), std::abs(dstdy[0]));
Float dt = std::max(std::abs(dstdx[1]), std::abs(dstdy[1]));
Float s0 = st[0] - ds, s1 = st[0] + ds;
Float t0 = st[1] - dt, t1 = st[1] + dt;
if (std::floor(s0) == std::floor(s1) &&
    std::floor(t0) == std::floor(t1)) {
    {Point sample Checkerboard2DTexture 643}
}
```

Otherwise, the lookup method approximates the filtered value by first computing a floating-point value that indicates what fraction of the filter region covers each of the two check types. This is equivalent to computing the average of the 2D step function that takes on the value 0 when we are in tex1 and 1 when we are in tex2 , over the filter region. Figure 10.21(a) shows a graph of the checkerboard function $c(x)$, defined as

$$c(x) = \begin{cases} 0 & \lfloor x \rfloor \text{ is even} \\ 1 & \text{otherwise.} \end{cases}$$

Given the average value, we can blend between the two subtextures, according to what fraction of the filter region each one is visible for.

The integral of the 1D checkerboard function $c(x)$ can be used to compute the average value of the function over some extent. Inspection of the graph reveals that

$$\int_0^x c(x) dx = \lfloor x/2 \rfloor + 2 \max(x/2 - \lfloor x/2 \rfloor - .5, 0).$$

float 1062

To compute the average value of the step function in two dimensions, we separately compute the integral of the checkerboard in each 1D direction in order to compute its average value over the filter region.

```
(Apply box filter to checkerboard region) ≡ 644
auto bumpInt = [](Float x) {
    return (int)std::floor(x / 2) +
        2 * std::max(x / 2 - (int)std::floor(x / 2) - (Float)0.5,
                      (Float)0); };

Float sint = (bumpInt(s1) - bumpInt(s0)) / (2 * ds);
Float tint = (bumpInt(t1) - bumpInt(t0)) / (2 * dt);
Float area2 = sint + tint - 2 * sint * tint;
if (ds > 1 || dt > 1)
    area2 = .5f;
return (1 - area2) * tex1->Evaluate(si) +
    area2 * tex2->Evaluate(si);
```

10.5.3 SOLID CHECKERBOARD

The Checkerboard2DTexture class from the previous section wraps a checkerboard pattern *around* the object in parameter space. We can also define a solid checkerboard pattern based on 3D texture coordinates so that the object appears carved out of 3D checker cubes (Figure 10.22). Like the 2D variant, this implementation chooses between texture functions based on the lookup position. Note that these two textures need not be solid textures themselves; the Checkerboard3DTexture merely chooses between them based on the 3D position of the point.



Figure 10.22: Dragon Model, Textured with the Checkerboard3DTexture Procedural Texture.
Notice how the model appears to be carved out of 3D checks, rather than having them pasted on its surface.

Checkerboard2DTexture 642

Checkerboard2DTexture::tex1
643

Checkerboard3DTexture 647

Float 1062

Texture::Evaluate() 615

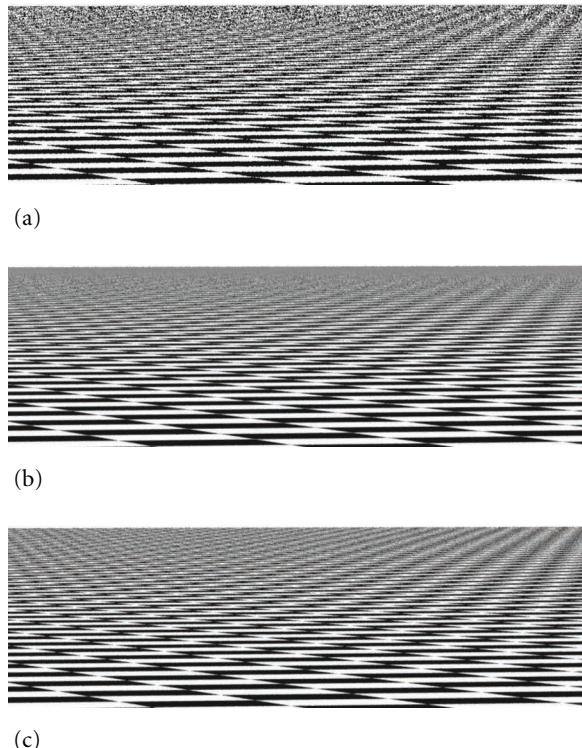


Figure 10.23: Comparisons of three approaches for antialiasing procedural textures, applied to the checkerboard texture. (a) No effort has been made to remove high-frequency variation from the texture function, so there are severe artifacts in the image, rendered with one sample per pixel. (b) This image shows the approach based on computing the filter region in texture space and averaging the texture function over that area, also rendered with one sample per pixel. (c) Here the checkerboard function was effectively supersampled by taking 16 samples per pixel and then point-sampling the texture. Both the area-averaging and the supersampling approaches give substantially better results than the first approach. In this example, supersampling gives the best results, since the averaging approach has blurred out the checkerboard pattern sooner than was needed because it approximates the filter region with its axis-aligned box.

```

<CheckerboardTexture Declarations> +≡
template <typename T> class Checkerboard3DTexture : public Texture<T> {
public:
    <Checkerboard3DTexture Public Methods 647>
private:
    <Checkerboard3DTexture Private Data 648>
};

<Checkerboard3DTexture Public Methods> ≡
    Checkerboard3DTexture(std::unique_ptr<TextureMapping3D> mapping,
        const std::shared_ptr<Texture<T>> &tex1,
        const std::shared_ptr<Texture<T>> &tex2)
        : mapping(std::move(mapping)), tex1(tex1), tex2(tex2) { }

```

```
(Checkerboard3DTexture Private Data) ≡ 647
    std::unique_ptr<TextureMapping3D> mapping;
    std::shared_ptr<Texture<T>> tex1, tex2;
```

Ignoring antialiasing, the basic computation to see if a point p is inside a 3D checker region is

$$(\lfloor p_x \rfloor + \lfloor p_y \rfloor + \lfloor p_z \rfloor) \bmod 2 = 0.$$

The Checkerboard3DTexture doesn't have any built-in support for antialiasing, so its implementation is fairly short.

```
(Checkerboard3DTexture Public Methods) +≡ 647
    T Evaluate(const SurfaceInteraction &si) const {
        Vector3f dpdx, dpdy;
        Point3f p = mapping->Map(si, &dpdx, &dpdy);
        if (((int)std::floor(p.x) + (int)std::floor(p.y) +
            (int)std::floor(p.z)) % 2 == 0)
            return tex1->Evaluate(si);
        else
            return tex2->Evaluate(si);
    }
```

10.6 NOISE

In order to write solid textures that describe complex surface appearances, it is helpful to be able to introduce some controlled variation to the process. Consider a wood floor made of individual planks; each plank's color is likely to be slightly different from the others. Or consider a windswept lake; we might want to have waves of similar amplitude across the entire lake, but we don't want them to be homogeneous over all parts of the lake (as they might be if they were constructed from a sum of sine waves, for example). Modeling this sort of variation in a texture helps make the final result look more realistic.

One difficulty in developing textures like these is that the renderer evaluates the surface's texture functions at an irregularly distributed set of points, where each evaluation is completely independent of the others. As such, procedural textures must *implicitly* define a complex pattern by answering queries about what the pattern's value is at all of these points. In contrast, the *explicit* pattern description approach is embodied by the PostScript® language, for example, which describes graphics on a page with a series of drawing commands. One difficulty that the implicit approach introduces is that the texture can't just call `RNG::UniformFloat()` at each point at which it is evaluated to introduce randomness: because each point would have a completely different random value from its neighbors, no coherence would be possible in the generated pattern.

An elegant way to address this issue of introducing controlled randomness to procedural textures in graphics is the application of what is known as a *noise function*. In general, noise functions used in graphics are smoothly varying functions taking $\mathbb{R}^n \rightarrow [-1, 1]$, for at least $n = 1, 2, 3$, without obvious repetition. One of the most crucial properties of a practical noise function is that it be band limited with a known maximum frequency.

```
Checkerboard3DTexture:: 648
    mapping
    648
    Checkerboard3DTexture::tex1
    648
    Checkerboard3DTexture::tex2
    648
    Point3f 68
    RNG::UniformFloat() 1066
    SurfaceInteraction 116
    Texture 614
    Texture::Evaluate() 615
    TextureMapping3D 614
    TextureMapping3D::Map() 614
    Vector3f 60
```

This makes it possible to control the frequency content added to a texture due to the noise function so that frequencies higher than those allowed by the Nyquist limit aren't introduced.

Many of the noise functions that have been developed are built on the idea of an integer lattice over \mathbb{R}^3 . First, a value is associated with each integer (x, y, z) position in space. Then, given an arbitrary position in space, the eight surrounding lattice values are found. These lattice values are then interpolated to compute the noise value at the particular point. This idea can be generalized or restricted to more or fewer dimensions d , where the number of lattice points is 2^d . A simple example of this approach is *value noise*, where pseudo-random numbers between -1 and 1 are associated with each lattice point, and actual noise values are computed with trilinear interpolation or with a more complex spline interpolant, which can give a smoother result by avoiding derivative discontinuities when moving from one lattice cell to another.

For such a noise function, given an integer (x, y, z) lattice point, it must be possible to efficiently compute its parameter value in a way that always associates the same value with each lattice point. Because it is infeasible to store values for all possible (x, y, z) points, some compact representation is needed. One option is to use a hash function, where the coordinates are hashed and then used to look up parameters from a fixed-size table of precomputed pseudo-random parameter values.

10.6.1 PERLIN NOISE

In pbrt we will implement a noise function introduced by Ken Perlin (1985a, 2002); as such, it is known as *Perlin noise*. It has a value of zero at all (x, y, z) integer lattice points. Its variation comes from gradient vectors at each lattice point that guide the interpolation of a smooth function in between the points (Figure 10.24). This noise function has many of the desired characteristics of a noise function described above, is computationally efficient, and is easy to implement. Figure 10.25 shows its value rendered on a sphere.

```
{Texture Method Definitions} +≡
    Float Noise(Float x, Float y, Float z) {
        {Compute noise cell coordinates and offsets 649}
        {Compute gradient weights 651}
        {Compute trilinear interpolation of weights 653}
    }
```

For convenience, there is also a variant of `Noise()` that takes a `Point3f` directly:

```
{Texture Method Definitions} +≡
    Float Noise(const Point3f &p) { return Noise(p.x, p.y, p.z); }
```

The implementation first computes the integer coordinates of the cell that contains the given point and the fractional offsets of the point from the lower cell corner:

`Float` 1062
`Noise()` 649
`Point3f` 68

```
{Compute noise cell coordinates and offsets} ≡
    int ix = std::floor(x), iy = std::floor(y), iz = std::floor(z);
    float dx = x - ix, dy = y - iy, dz = z - iz;
```

649

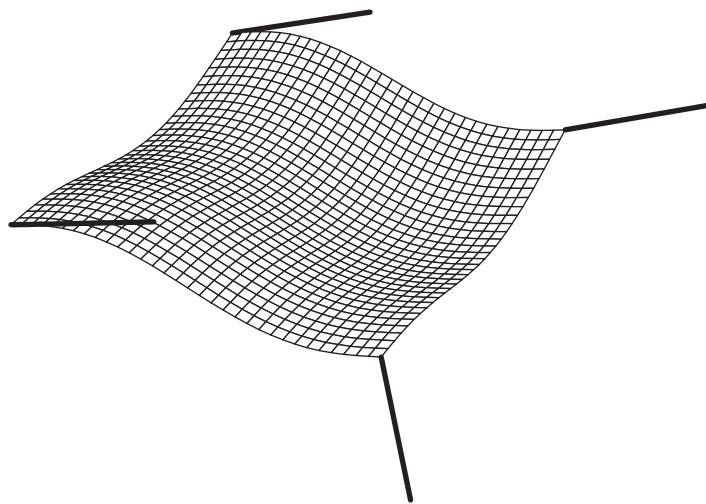


Figure 10.24: The Perlin noise function is computed by generating a smooth function that is zero but with a given derivative at integer lattice points. The derivatives are used to compute a smooth interpolating surface. Here, a 2D slice of the noise function is shown with four gradient vectors.

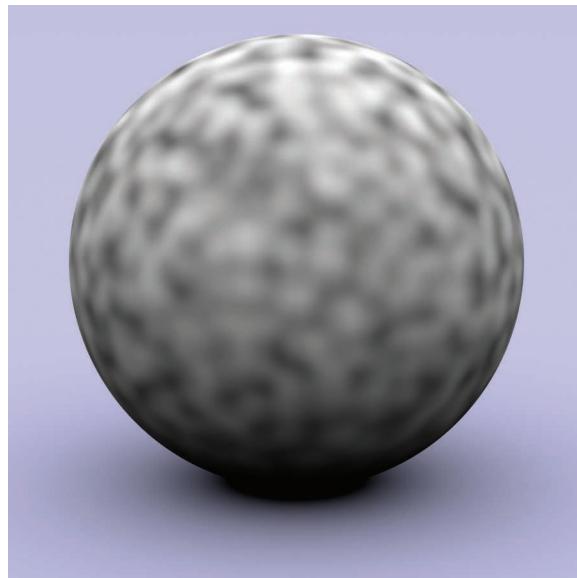


Figure 10.25: Perlin's Noise Function Modulating the Diffuse Color of a Sphere.

It next calls `Grad()` to get eight weight values, one for each corner of the cell that the point lies inside. `Grad()` uses the cell indices to index into a table; for efficiency we ensure that all of the indices are within the range of the table here by zeroing any high bits that would put a component past the table's size. (The table size must be a power of two for this trick to work—otherwise an expensive integer modulus operation would be needed in place of the bitwise “and.”)

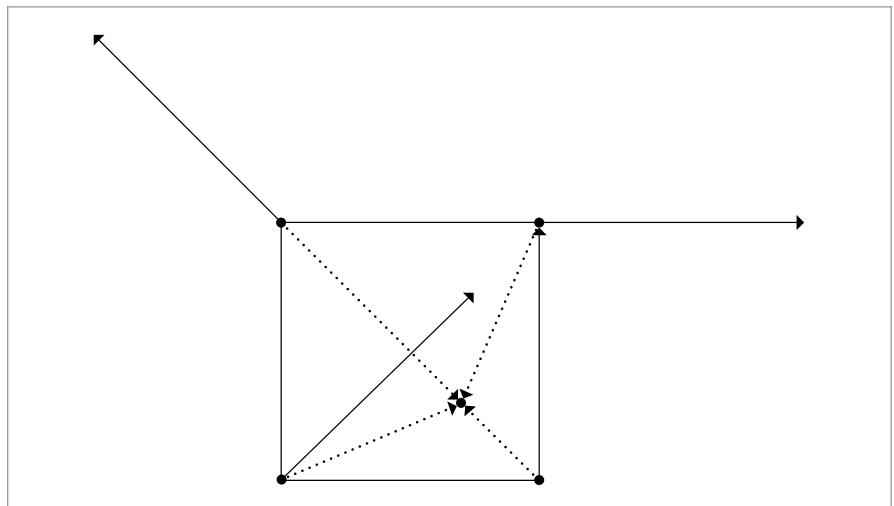
(Compute gradient weights) ≡ 649

```

ix &= NoisePermSize - 1;
iy &= NoisePermSize - 1;
iz &= NoisePermSize - 1;
Float w000 = Grad(ix,    iy,    iz,    dx,    dy,    dz);
Float w100 = Grad(ix+1, iy,    iz,    dx-1, dy,    dz);
Float w010 = Grad(ix,    iy+1, iz,    dx,    dy-1, dz);
Float w110 = Grad(ix+1, iy+1, iz,    dx-1, dy-1, dz);
Float w001 = Grad(ix,    iy,    iz+1, dx,    dy,    dz-1);
Float w101 = Grad(ix+1, iy,    iz+1, dx-1, dy,    dz-1);
Float w011 = Grad(ix,    iy+1, iz+1, dx,    dy-1, dz-1);
Float w111 = Grad(ix+1, iy+1, iz+1, dx-1, dy-1, dz-1);

```

Each integer lattice point has a gradient vector associated with it. The influence of the gradient vector for any point inside the cell is obtained by computing the dot product of the vector from the gradient’s corner to the lookup point and the gradient vector (Figure 10.26); this is handled by the `Grad()` function. Note that the vectors to the corners other than the lower left one can be easily computed incrementally based on that vector.



Float 1062
Grad() 652
NoisePermSize 652

Figure 10.26: The dot product of the vector from the corners of the cell to the lookup point (dotted lines) with each of the gradient vectors (solid lines) gives the influence of each gradient to the noise value at the point.

The gradient vector for a particular integer lattice point is found by indexing into a precomputed table of integer values, `NoisePerm`. The four low-order bits of the table's value for the lattice point determine which of 16 gradient vectors is associated with it. In a preprocessing step, this table of size `NoisePermSize` was filled with numbers from 0 to `NoisePermSize`-1 and then randomly permuted. These values were then duplicated, making an array of size `2*NoisePermSize` that holds the table twice in succession. The second copy of the table makes lookups in the following code slightly more efficient.

Given a particular (`ix`, `iy`, `iz`) lattice point, a series of table lookups gives a value from the random-number table:

```
NoisePerm[NoisePerm[NoisePerm[ix] + iy] + iz];
```

By doing three nested permutations in this way, rather than `NoisePerm[ix+iy+iz]`, for example, the final result is more irregular. The first approach doesn't usually return the same value if `ix` and `iy` are interchanged, as the second does. Furthermore, since the table was replicated to be twice the original length, the lookups can be done as described above, eliminating the need for modulus operations in code along the lines of

```
(NoisePerm[ix] + iy) % NoisePermSize
```

Given a final value from the permutation table that determines the gradient number, the dot product with the corresponding gradient vector must be computed. However, the gradient vectors do not need to be represented explicitly. All of the gradients use only -1, 0, or 1 in their coordinates, so that the dot products reduce to addition of some (possibly negated) components of the vector.⁵ The final implementation is the following:

(Texture Method Definitions) \equiv

```
inline Float Grad(int x, int y, int z, Float dx, Float dy, Float dz) {
    int h = NoisePerm[NoisePerm[NoisePerm[x] + y] + z];
    h &= 15;
    Float u = h < 8 || h == 12 || h == 13 ? dx : dy;
    Float v = h < 4 || h == 12 || h == 13 ? dy : dz;
    return ((h & 1) ? -u : u) + ((h & 2) ? -v : v);
}
```

(Perlin Noise Data) \equiv

```
static constexpr int NoisePermSize = 256;
static int NoisePerm[2 * NoisePermSize] = {
    151, 160, 137, 91, 90, 15, 131, 13, 201, 95, 96,
    53, 194, 233, 7, 225, 140, 36, 103, 30, 69, 142,
    (Remainder of the noise permutation table)
};
```

`Float` [1062](#)

`NoisePerm` [652](#)

`NoisePermSize` [652](#)

⁵ The original formulation of Perlin noise also had a precomputed table of pseudo-random gradient directions, although Perlin has more recently suggested that the randomness from the permutation table is enough to remove regularity from the noise function.

Given these eight contributions from the gradients, the next step is to trilinearly interpolate between them at the point. Rather than interpolating with dx , dy , and dz directly, though, each of these values is passed through a smoothing function. This ensures that the noise function has first- and second-derivative continuity as lookup points move between lattice cells.

```
(Texture Method Definitions) +≡
    inline Float NoiseWeight(Float t) {
        Float t3 = t * t * t;
        Float t4 = t3 * t;
        return 6 * t4 * t - 15 * t4 + 10 * t3;
    }
```

(Compute trilinear interpolation of weights) ≡

```
Float wx = NoiseWeight(dx), wy = NoiseWeight(dy), wz = NoiseWeight(dz);
Float x00 = Lerp(wx, w000, w100);
Float x10 = Lerp(wx, w010, w110);
Float x01 = Lerp(wx, w001, w101);
Float x11 = Lerp(wx, w011, w111);
Float y0 = Lerp(wy, x00, x10);
Float y1 = Lerp(wy, x01, x11);
return Lerp(wz, y0, y1);
```

649

10.6.2 RANDOM POLKA DOTS

A basic use of the noise function is as part of a polka dot texture that divides (s, t) texture space into rectangular cells (Figure 10.27). Each cell has a 50% chance of having a dot inside of it, and the dots are randomly placed inside their cells. `DotsTexture` takes the usual 2D mapping function, as well as two `Textures`, one for the regions of the surface outside of the dots and one for the regions inside. It is defined in the files `textures/dots.h` and `textures/dots.cpp`.

(DotsTexture Declarations) ≡

```
template <typename T> class DotsTexture : public Texture<T> {
public:
    (DotsTexture Public Methods 653)
private:
    (DotsTexture Private Data 653)
};
```

DotsTexture 653
 DotsTexture::insideDot 653
 DotsTexture::mapping 653
 DotsTexture::outsideDot 653
 Float 1062
 Lerp() 1079
 NoiseWeight() 653
 Texture 614
 TextureMapping2D 609

(DotsTexture Public Methods) ≡

```
DotsTexture(std::unique_ptr<TextureMapping2D> mapping,
           const std::shared_ptr<Texture<T>> &outsideDot,
           const std::shared_ptr<Texture<T>> &insideDot)
    : mapping(std::move(mapping)), outsideDot(outsideDot),
      insideDot(insideDot) { }
```

653

(DotsTexture Private Data) ≡

```
std::unique_ptr<TextureMapping2D> mapping;
std::shared_ptr<Texture<T>> outsideDot, insideDot;
```

653

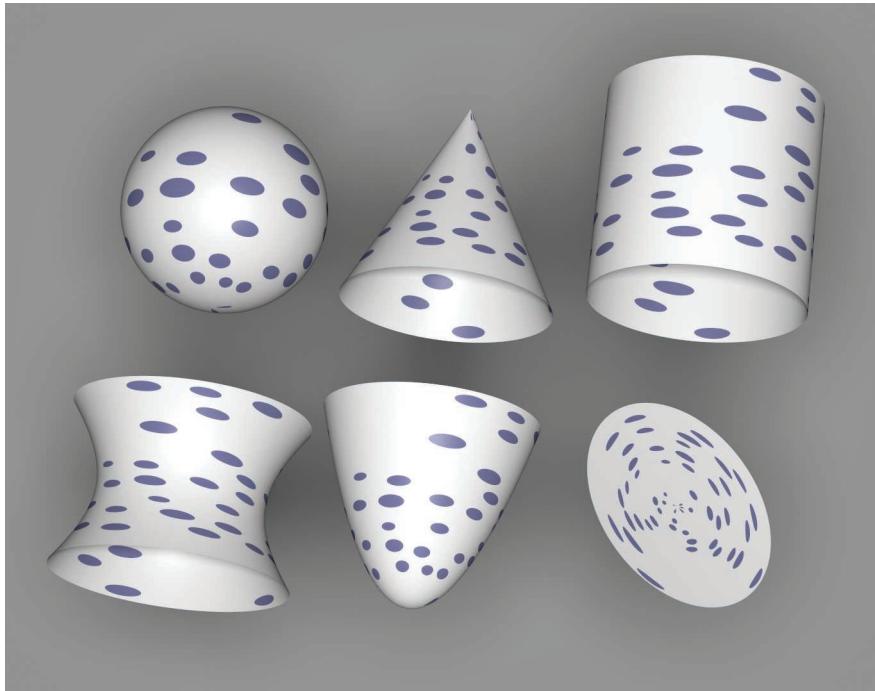


Figure 10.27: The Polka Dot Texture Applied to All of pbrt’s Quadric Shapes.

The evaluation function starts by taking the (s, t) texture coordinates and computing integer $sCell$ and $tCell$ values, which give the coordinates of the cell they are inside. We will not consider antialiasing of the polka dots texture here; an exercise at the end of the chapter outlines how this might be done.

(DotsTexture Public Methods) +≡

653

```
T Evaluate(const SurfaceInteraction &si) const {
    (Compute cell indices for dots 654)
    (Return insideDot result if point is inside dot 655)
    return outsideDot->Evaluate(si);
}
```

(Compute cell indices for dots) ≡

654

```
Vector2f dstdx, dstdy;
Point2f st = mapping->Map(si, &dstdx, &dstdy);
int sCell = std::floor(st[0] + .5f), tCell = std::floor(st[1] + .5f);
```

Once the cell coordinate is known, it’s necessary to decide if there is a polka dot in the cell. Obviously, this computation needs to be consistent so that for each time this routine runs for points in a particular cell it returns the same result. Yet we’d like the result not to be completely regular (e.g., with a dot in every other cell). Noise solves this problem: by evaluating the noise function at a position that is the same for all points inside this cell—

DotsTexture::mapping 653
 DotsTexture::outsideDot 653
 Point2f 68
 SurfaceInteraction 116
 Texture::Evaluate() 615
 TextureMapping2D::Map() 610
 Vector2f 60

$(sCell+.5, tCell+.5)$ —we can compute an irregularly varying but consistent value for each cell.⁶ If this value is greater than zero, a dot is placed in the cell.

If there is a dot in the cell, the noise function is used again to randomly shift the center of the dot within the cell. The points at which the noise function is evaluated for the center shift are offset by arbitrary constant amounts, however, so that noise values from different noise cells are used from them, eliminating a possible source of correlation with the noise value used to determine the presence of a dot in the first place. (Note that the dot’s radius must be small enough so that it doesn’t spill over the cell’s boundary after being shifted; in that case, points where the texture was being evaluated would also need to consider the dots based in neighboring cells as potentially affecting their texture value.)

Given the dot center and radius, the texture needs to decide if the (s, t) coordinates are within the radius of the shifted center. It does this by computing their squared distance to the center and comparing it to the squared radius.

```

(Return insideDot result if point is inside dot) ≡ 654
    if (Noise(sCell + .5f, tCell + .5f) > 0) {
        Float radius = .35f;
        Float maxShift = 0.5f - radius;
        Float sCenter = sCell +
            maxShift * Noise(sCell + 1.5f, tCell + 2.8f);
        Float tCenter = tCell +
            maxShift * Noise(sCell + 4.5f, tCell + 9.8f);
        Vector2f dst = st - Point2f(sCenter, tCenter);
        if (dst.LengthSquared() < radius*radius)
            return insideDot->Evaluate(si);
    }

```

10.6.3 NOISE IDIOMS AND SPECTRAL SYNTHESIS

The fact that noise is a band-limited function means that its frequency content can be adjusted by scaling the domain over which it is evaluated. For example, if $\text{Noise}(p)$ has some known frequency content, then the frequency content of $\text{Noise}(2*p)$ will be twice as high. This is just like the relationship between the frequency content of $\sin(x)$ and $\sin(2x)$. This technique can be used to create a noise function with a desired rate of variation.

For many applications in procedural texturing, it’s useful to have variation over multiple scales—for example, to add finer variations to the base noise function. One effective way to do this with noise is to compute patterns via *spectral synthesis*, where a complex function $f_s(s)$ is defined by a sum of contributions from another function $f(x)$:

```

DotsTexture::insideDot 653
Float 1062
Noise() 649
Point2f 68
Texture::Evaluate() 615
Vector2f 60

```

$$f_s(x) = \sum_i w_i f(s_i x),$$

6 Recall that the noise function always returns zero at integer (x, y, z) coordinates, so we don’t want to just evaluate it at $(sCell, tCell)$. Although the 3D noise function would actually be evaluating noise at $(sCell, tCell, .5)$, slices through noise with integer values for any of the coordinates are not as well distributed as with all of them offset from the integers.

for a set of weight values w_i and parameter scale values s_i . If the base function $f(x)$ has a well-defined frequency content (e.g., is a sine or cosine function or a noise function), then each term $f(s_i x)$ also has a well-defined frequency content as described earlier. Because each term of the sum is weighted by a weight value w_i , the result is a sum of contributions of various frequencies, with different frequency ranges weighted differently.

Typically, the scales s_i are chosen in a geometric progression such that $s_i = 2s_{i-1}$ and the weights are $w_i = w_{i-1}/2$. The result is that as higher frequency variation is added to the function, it has relatively less influence on the overall shape of $f_s(x)$. Each additional term is called an *octave* of noise, since it has twice the frequency content of the previous one. When this scheme is used with Perlin noise, the result is often referred to as *fractional Brownian motion* (fBm), after a particular type of random process that varies in a similar manner.

Fractional Brownian motion is a useful building block for procedural textures because it gives a function with more complex variation than plain noise, while still being easy to compute and still having well-defined frequency content. The utility function `FBm()` implements the fractional Brownian motion function. Figure 10.28 shows two graphs of it.

In addition to the point at which to evaluate the function and the function's partial derivatives at that point, the function takes an `omega` parameter, which ranges from zero to one and affects the smoothness of the pattern by controlling the falloff of contributions at higher frequencies (values around 0.5 work well), and `maxOctaves`, which gives the maximum number of octaves of noise that should be used in computing the sum.

```
(Texture Method Definitions) +≡
Float FBm(const Point3f &p, const Vector3f &dpdx, const Vector3f &dpdy,
          Float omega, int maxOctaves) {
    (Compute number of octaves for antialiased FBm 658)
    (Compute sum of octaves of noise for FBm 658)
    return sum;
}
```

The implementation here uses a technique called *clamping* to antialias the fBm function. The idea is that when we are computing a value based on a sum of components, each with known frequency content, we should stop adding in components that would have frequencies beyond the Nyquist limit and instead add their average values to the sum. Because the average value of `Noise()` is zero, all that needs to be done is to compute the number of octaves such that none of the terms has excessively high frequencies and not evaluate the noise function for any higher octaves.

`Noise()` (and thus the first term of $f_s(x)$ as well) has a maximum frequency content of roughly $\omega = 1$. Each subsequent term represents a doubling of frequency content. Therefore, we would like to find the appropriate number of terms n such that if the sampling rate in noise space is s , then

$$\frac{s}{2^n} = 2,$$

`FBm()` 656

`Float` 1062

`Noise()` 649

`Point3f` 68

`Vector3f` 60

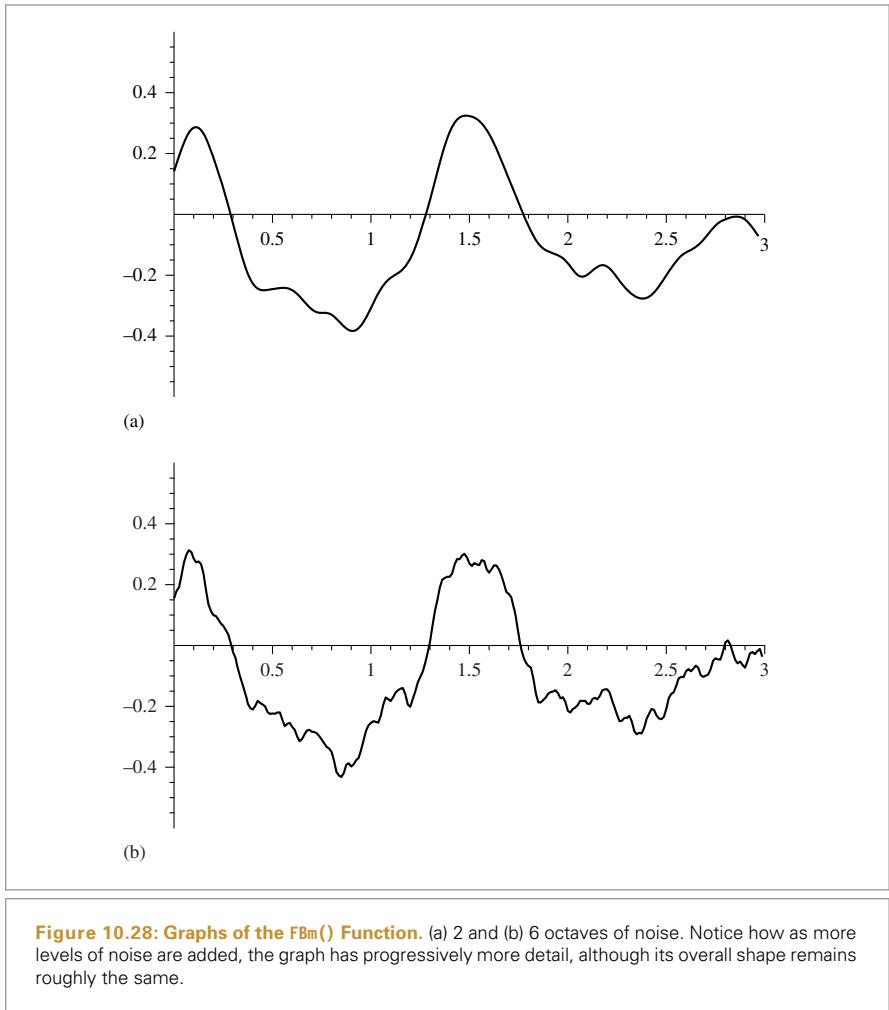


Figure 10.28: Graphs of the FBm() Function. (a) 2 and (b) 6 octaves of noise. Notice how as more levels of noise are added, the graph has progressively more detail, although its overall shape remains roughly the same.

which ensures that there are frequencies right up to the Nyquist frequency but not past it. Equivalently,

$$\begin{aligned} 2^{n+1} &= s \\ n + 1 &= \log_2 s \\ n &= -1 + \log_2 s \\ n &= -1 + \frac{1}{2} \log_2 s^2, \end{aligned}$$

where we've used the identity $\log_2 x = 1/n \log_2 x^n$ to write the last expression in a more convenient form for the following.

FBm() 656

The squared sampling rate s^2 can be computed with one over the maximum of the squared length of the differentials $\partial p/\partial x$ and $\partial p/\partial y$, which we'll denote by l^2 . We can

turn this inversion into a negation of the logarithm, and equivalently write this as:

$$n = -1 - \frac{1}{2} \log_2 l^2.$$

(Compute number of octaves for antialiased FBm) ≡

656, 660

```
Float len2 = std::max(dpx.LengthSquared(), dpy.LengthSquared());
Float n = Clamp(-1 - .5f * Log2(len2), 0, maxOctaves);
int nInt = std::floor(n);
```

Finally, the integral number of octaves up to the Nyquist limit are added together and the last octave is faded in, according to the fractional part of n. This ensures that successive octaves of noise fade in gradually, rather than appearing abruptly, which can cause visually noticeable artifacts at the transitions. The implementation here actually increases the frequency between octaves by 1.99, rather than by a factor of 2, in order to reduce the impact of the fact that the noise function is zero at integer lattice points. This breaks up that regularity across sums of octaves of noise, which can also lead to subtle visual artifacts.

(Compute sum of octaves of noise for FBm) ≡

656

```
Float sum = 0, lambda = 1, o = 1;
for (int i = 0; i < nInt; ++i) {
    sum += o * Noise(lambda * p);
    lambda *= 1.99f;
    o *= omega;
}
Float nPartial = n - nInt;
sum += o * SmoothStep(.3f, .7f, nPartial) * Noise(lambda * p);
```

The `SmoothStep()` function takes a minimum and maximum value and a point at which to evaluate a smooth interpolating function. If the point is below the minimum zero is returned, and if it's above the maximum one is returned. Otherwise, it smoothly interpolates between zero and one using a cubic Hermite spline.

(Texture Inline Functions) ≡

```
inline Float SmoothStep(Float min, Float max, Float value) {
    Float v = Clamp((value - min) / (max - min), 0, 1);
    return v * v * (-2 * v + 3);
}
```

Closely related to the `FBm()` function is the `Turbulence()` function. It also computes a sum of terms of the noise function but takes the absolute value of each one:

$$f_s(x) = \sum_i w_i |f(s_i x)|.$$

Taking the absolute value introduces first-derivative discontinuities in the synthesized function, and thus the turbulence function has infinite frequency content. Nevertheless, the visual characteristics of this function can be quite useful for procedural textures. Figure 10.29 shows two graphs of the turbulence function.

Clamp() 1062
FBm() 656
Float 1062
Log2() 1063
Noise() 649
SmoothStep() 658
Vector3::LengthSquared() 65

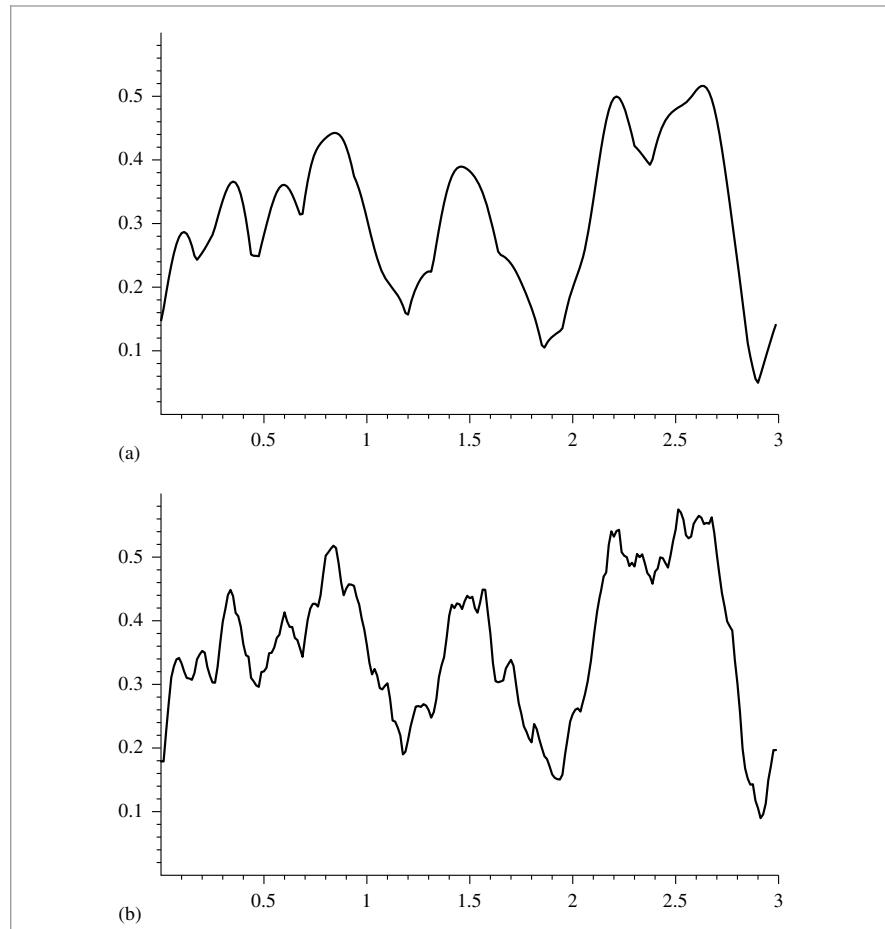


Figure 10.29: Graphs of the Turbulence() function for (a) 2 and (b) 6 octaves of noise. Note that the first derivative discontinuities introduced by taking the absolute value of the noise function make this function substantially rougher than FBm.

The Turbulence() implementation here tries to antialias itself in the same way that FBm() did. As described earlier, however, the first-derivative discontinuities in the function introduce infinitely high-frequency content, so these efforts can't hope to be perfectly successful. The Turbulence() antialiasing here at least eliminates some of the worst of the artifacts; otherwise, increasing the pixel sampling rate is the best recourse. In practice, this function doesn't alias too terribly when used in procedural textures, particularly compared to the aliasing from infinitely high frequencies from geometric and shadow edges.

FBm() 656

Turbulence() 660

(Texture Method Definitions) +≡

```
Float Turbulence(const Point3f &p, const Vector3f &dpdx,
    const Vector3f &dpdy, Float omega, int maxOctaves) {
    (Compute number of octaves for antialiased FBm 658)
    (Compute sum of octaves of noise for turbulence 660)
    (Account for contributions of clamped octaves in turbulence 660)
    return sum;
}
```

(Compute sum of octaves of noise for turbulence) ≡ 660

```
Float sum = 0, lambda = 1, o = 1;
for (int i = 0; i < nInt; ++i) {
    sum += o * std::abs(Noise(lambda * p));
    lambda *= 1.99f;
    o *= omega;
}
```

The average value of the absolute value of the noise function is roughly 0.2; this value should be added to the sum for the octaves where the noise function's estimated frequency would be higher than the sampling rate.

(Account for contributions of clamped octaves in turbulence) ≡ 660

```
Float nPartial = n - nInt;
sum += o * Lerp(SmoothStep(.3f, .7f, nPartial),
    0.2, std::abs(Noise(lambda * p)));
for (int i = nInt; i < maxOctaves; ++i) {
    sum += o * 0.2f;
    o *= omega;
}
```

10.6.4 BUMPY AND WRINKLED TEXTURES

The fBm and turbulence functions are particularly useful as a source of random variation for bump mapping. The `FBmTexture` is a `Float`-valued texture that uses `FBm()` to compute offsets, and `WrinkledTexture` uses `Turbulence()` to do so. They are demonstrated in Figures 10.30 and 10.31 and are implemented in `textures/fbm.h`, `textures/fbm.cpp`, `textures/wrinkled.h`, and `textures/wrinkled.cpp`.

(FBmTexture Declarations) ≡

```
template <typename T> class FBmTexture : public Texture<T> {
public:
    (FBmTexture Public Methods 662)
private:
    std::unique_ptr<TextureMapping3D> mapping;
    const Float omega;
    const int octaves;
};
```

`FBm()` 656
`FBmTexture` 660
`Float` 1062
`Lerp()` 1079
`Noise()` 649
`Point3f` 68
`SmoothStep()` 658
`Texture` 614
`TextureMapping3D` 614
`Turbulence()` 660
`Vector3f` 60
`WrinkledTexture` 662



Figure 10.30: Sphere with FBmTexture Used for Bump Mapping.

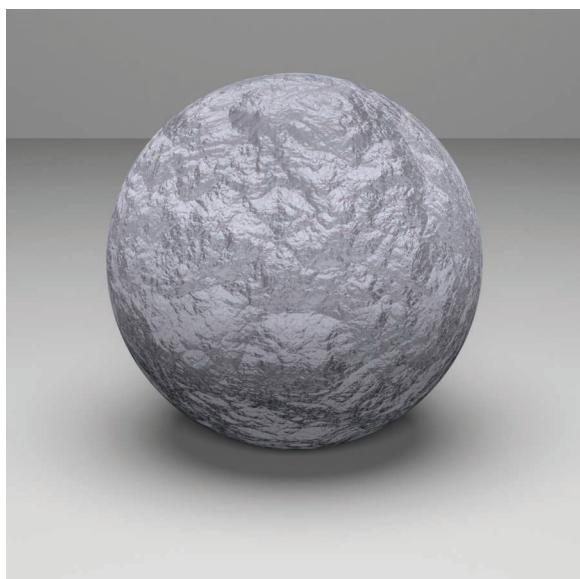


Figure 10.31: WrinkledTexture Used as Bump Mapping Function for Sphere.

FBmTexture [660](#)

WrinkledTexture [662](#)

```
(FBmTexture Public Methods) ≡ 660
FBmTexture(std::unique_ptr<TextureMapping3D> mapping, int octaves,
           float omega)
: mapping(std::move(mapping)), omega(omega), octaves(octaves) { }
```

```
(FBmTexture Public Methods) +≡ 660
T Evaluate(const SurfaceInteraction &si) const {
    Vector3f dpdx, dpdy;
    Point3f P = mapping->Map(si, &dpdx, &dpdy);
    return FBm(P, dpdx, dpdy, omega, octaves);
}
```

The implementation of WrinkledTexture is almost identical to FBmTexture, save for a call to Turbulence() instead of FBm(). As such, it isn't included here.

10.6.5 WINDY WAVES

Application of fBm can give a reasonably convincing representation of waves (Ebert et al. 2003). Figures 1.11 and 4.1 use this texture for the water in those scenes. This Texture is based on two observations. First, across the surface of a wind-swept lake (for example), some areas are relatively smooth and some are more choppy; this effect comes from the natural variation of the wind's strength from area to area. Second, the overall form of individual waves on the surface can be described well by the fBm-based wave pattern scaled by the wind strength. This texture is implemented in `textures/windy.h` and `textures/windy.cpp`.

```
(WindyTexture Declarations) ≡
template <typename T> class WindyTexture : public Texture<T> {
public:
    (WindyTexture Public Methods 662)
private:
    std::unique_ptr<TextureMapping3D> mapping;
};
```

```
(WindyTexture Public Methods) ≡ 662
WindyTexture(std::unique_ptr<TextureMapping3D> mapping)
: mapping(std::move(mapping)) { }
```

The evaluation function uses two calls to the FBm() function. The first scales down the point P by a factor of 10; as a result, the first call to FBm() returns relatively low-frequency variation over the surface of the object being shaded. This value is used to determine the local strength of the wind. The second call determines the amplitude of the wave at the particular point, independent of the amount of wind there. The product of these two values gives the actual wave offset for the particular location.

FBm() 656
FBmTexture 660
FBmTexture::mapping 660
FBmTexture::octaves 660
FBmTexture::omega 660
float 1062
Point3f 68
SurfaceInteraction 116
Texture 614
TextureMapping3D 614
TextureMapping3D::Map() 614
Turbulence() 660
Vector3f 60
WindyTexture 662
WrinkledTexture 662

```

⟨WindyTexture Public Methods⟩ +≡
    T Evaluate(const SurfaceInteraction &si) const {
        Vector3f dpdx, dpdy;
        Point3f P = mapping->Map(si, &dpdx, &dpdy);
        Float windStrength = FBm(.1f * P, .1f * dpdx, .1f * dpdy, .5, 3);
        Float waveHeight = FBm(P, dpdx, dpdy, .5, 6);
        return std::abs(windStrength) * waveHeight;
    }

```

662

10.6.6 MARBLE

Another classic use of the noise function is to perturb texture coordinates before using another texture or lookup table. For example, a facsimile of marble can be made by modeling the marble material as a series of layered strata and then using noise to perturb the coordinate used for finding a value among the strata. The `MarbleTexture` in this section implements this approach. Figure 10.32 illustrates the idea behind this texture. On the left, the layers of marble are indexed directly using the `y` coordinate of the point on the sphere. On the right, `fBm` has been used to perturb the `y` value, introducing variation. This texture is implemented in `textures/marble.h` and `textures/marble.cpp`.

```

⟨MarbleTexture Declarations⟩ ≡
    class MarbleTexture : public Texture<Spectrum> {
    public:
        ⟨MarbleTexture Public Methods 664⟩
    private:
        ⟨MarbleTexture Private Data 664⟩
    };

```

`FBm()` 656
`Float` 1062
`MarbleTexture` 663
`Point3f` 68
`Spectrum` 315
`SurfaceInteraction` 116
`Texture` 614
`TextureMapping3D::Map()` 614
`Vector3f` 60
`WindyTexture::mapping` 662

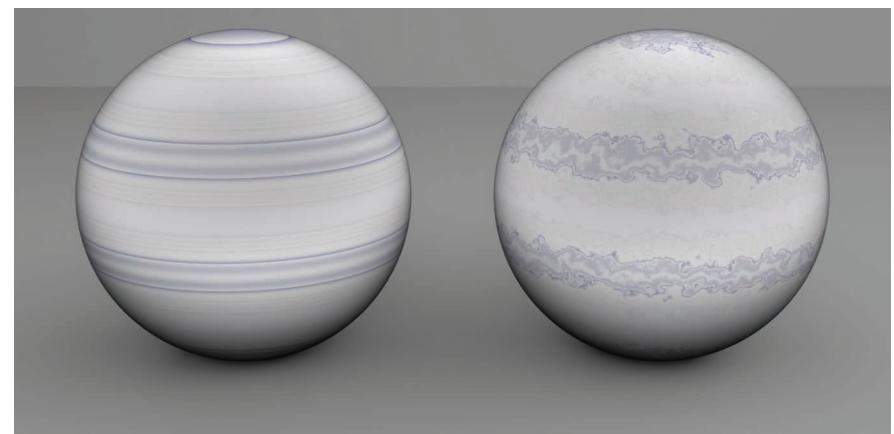


Figure 10.32: Marble. The `MarbleTexture` perturbs the coordinate used to index into a 1D table of colors using `FBm`, giving a plausible marble appearance.

The texture takes the usual set of parameters to control the `FBm()` function that will be used to perturb the lookup coordinate. The `variation` parameter modulates the magnitude of the perturbation.

(MarbleTexture Public Methods) ≡ 663

```
MarbleTexture(std::unique_ptr<TextureMapping3D> mapping, int octaves,
             Float omega, Float scale, Float variation)
: mapping(std::move(mapping)), octaves(octaves), omega(omega),
  scale(scale), variation(variation) {}
```

(MarbleTexture Private Data) ≡ 663

```
std::unique_ptr<TextureMapping3D> mapping;
const int octaves;
const Float omega, scale, variation;
```

An offset into the marble layers is computed by adding the variation to the point's `y` component and using the sine function to remap its value into the range [0, 1]. The *(Evaluate marble spline at t)* fragment uses the `t` value as the evaluation point for a cubic spline through a series of colors that are similar to those of real marble.

(MarbleTexture Public Methods) +≡ 663

```
Spectrum Evaluate(const SurfaceInteraction &si) const {
    Vector3f dpdx, dpdy;
    Point3f p = mapping->Map(si, &dpdx, &dpdy);
    p *= scale;
    Float marble = p.y + variation *
        FBm(p, scale * dpdx, scale * dpdy, omega, octaves);
    Float t = .5f + .5f * std::sin(marble);
    (Evaluate marble spline at t)
}
```

FURTHER READING

The cone-tracing method of Amanatides (1984) was one of the first techniques for automatically estimating filter footprints for ray tracing. The beam-tracing algorithm of Heckbert and Hanrahan (1984) was another early extension of ray tracing to incorporate an area associated with each image sample rather than just an infinitesimal ray. The pencil-tracing method of Shinya, Takahashi, and Naito (1987) is another approach to this problem. Other related work on the topic of associating areas or footprints with rays includes Mitchell and Hanrahan's paper (1992) on rendering caustics and Turkowski's technical report (1993).

Collins (1994) estimated the ray footprint by keeping a tree of all rays traced from a given camera ray, examining corresponding rays at the same level and position. The ray differentials used in pbrt are based on Igehy's (1999) formulation, which was extended by Suykens and Willemans (2001) to handle glossy reflection in addition to perfect specular reflection. Worley's chapter in *Texturing and Modeling* (Ebert et al. 2003) on computing differentials for filter regions presents an approach similar to ours. See Elek et al. (2014)

`FBm()` [656](#)
`Float` [1062](#)
`MarbleTexture` [663](#)
`MarbleTexture::mapping` [664](#)
`MarbleTexture::octaves` [664](#)
`MarbleTexture::omega` [664](#)
`MarbleTexture::scale` [664](#)
`MarbleTexture::variation` [664](#)
`Point3f` [68](#)
`Spectrum` [315](#)
`SurfaceInteraction` [116](#)
`TextureMapping3D` [614](#)
`TextureMapping3D::Map()` [614](#)
`Vector3f` [60](#)

for an extension of ray differentials to include wavelength, which can improve results with full-spectral rendering.

Two-dimensional texture mapping with images was first introduced to graphics by Blinn and Newell (1976). Ever since Crow (1977) identified aliasing as the source of many errors in images in graphics, quite a bit of work has been done to find efficient and effective ways of antialiasing image maps. Dungan, Stenger, and Sutty (1978) were the first to suggest creating a pyramid of prefiltered texture images; they used the nearest texture sample at the appropriate level when looking up texture values, using supersampling in screen space to antialias the result. Feibusch, Levoy, and Cook (1980) investigated a spatially varying filter function, rather than a simple box filter. (Blinn and Newell were aware of Crow's results and used a box filter for their textures.)

Williams (1983) used a MIP map image pyramid for texture filtering with trilinear interpolation. Shortly thereafter, Crow (1984) introduced summed area tables, which make it possible to efficiently filter over axis-aligned rectangular regions of texture space. Summed area tables handle anisotropy better than Williams's method, although only for primarily axis-aligned filter regions. Heckbert (1986) wrote a good general survey of texture mapping algorithms through the mid-1980s.

Greene and Heckbert (1986) originally developed the elliptically weighted average technique, and Heckbert's master's thesis (1989) put the method on a solid theoretical footing. Fournier and Fiume (1988) developed an even higher quality texture filtering method that focuses on using a bounded amount of computation per lookup. Nonetheless, their method appears to be less efficient than EWA overall. Lansdale's master's thesis (1991) has an extensive description of EWA and Fournier and Fiume's method, including implementation details.

More recently, a number of researchers have investigated generalizing Williams's original method using a series of trilinear MIP map samples in an effort to increase quality without having to pay the price for the general EWA algorithm. By taking multiple samples from the MIP map, anisotropy is handled well while preserving the computational efficiency. Examples include Barkans's (1997) description of texture filtering in the Talisman architecture, McCormack et al.'s (1999) Feline method, and Cant and Shrubsole's (2000) technique. Manson and Schaefer (2013, 2014) have recently shown how to accurately approximate a variety of filter functions with a fixed small number of bilinearly interpolated sample values. This approach is particularly useful on GPUs, where hardware-accelerated bilinear interpolation is available.

Gamma correction has a long history in computer graphics. Poynton (2002a, 2002b) has written comprehensive FAQs on issues related to color representation and gamma correction. Most modern displays are based on the sRGB color space, which has a gamma of roughly 2.2 (International Electrotechnical Commission (IEC) 1999). See Gritz and d'Eon (2008) for a detailed discussion of the implications of gamma correction for rendering and how to correctly account for it in rendering systems.

Smith's (2002) Web site and document on audio resampling gives a good overview of resampling signals in one dimension. Heckbert's (1989a) zoom source code is the canonical reference for image resampling. His implementation carefully avoids feedback without

using auxiliary storage, unlike ours in this chapter, which allocates additional temporary buffer space to do so.

Three-dimensional solid texturing was originally developed by Gardner (1984, 1985), Perlin (1985a), and Peachey (1985). Norton, Rockwood, and Skolmoski (1982) developed the *clamping* method that is widely used for antialiasing textures based on solid texturing. The general idea of procedural texturing was introduced by Cook (1984), Perlin (1985a), and Peachey (1985).

Peachey's chapter in *Texturing and Modeling* (Ebert et al. 2003) has a thorough summary of approaches to noise functions. After Perlin's original noise function, both Lewis (1989) and van Wijk (1991) developed alternatives that made different time/quality trade-offs. Worley (1996) has developed a quite different noise function for procedural texturing that is well suited for cellular and organic patterns. Perlin (2002) revised his noise function to correct a number of subtle shortcomings.

Noise functions have received additional attention from the research community in recent years. (Lagae et al. (2010) have a good survey of work up to that year.) Building on Lewis's observation that individual bands of Perlin's noise function actually have frequency content over a fairly wide range (Lewis 1989), Cook and DeRose (2005) also identified the problem that 2D slices through 3D noise functions aren't in general band limited, even if the original 3D noise function is. They proposed a new noise function that addresses both of these issues. Goldberg et al. (2008) developed a noise function that makes efficient anisotropic filtering possible, leading to higher quality results than just applying the clamping approach for antialiasing. Their method is also well suited to programmable graphics hardware. Kensler et al. (2008) suggested a number of improvements to Perlin's revised noise function.

Lagae et al. (2009) have developed a noise function that has good frequency control and can be mapped well to surfaces even without a surface parameterization. Lagae and Drettakis (2011) showed how to compute high quality anisotropically filtered values of this noise function. More recently, Galerne et al. (2012) showed how to automatically determine parameters to this noise function so that the result matches example images. Further work on this topic was done by Du et al. (2013) and Gilet et al. (2014).

The first languages and systems that supported the idea of user-supplied procedural shaders were developed by Cook (1984) and Perlin (1985a). (The texture composition model in this chapter is similar to Cook's shade trees.) The RenderMan shading language, described in a paper by Hanrahan and Lawson (1990), remains the classic shading language in graphics, though a more modern shading language is available in Open Shading Language (OSL) (Gritz et al. 2010), which is open source and increasingly used for production rendering. It follows the model of the shader returning a representation of the material rather than a final color value, like the approach introduced in Chapter 9. See also Karrenberg et al. (2010), who introduced the *AnySL* shading language, which was designed for both high performance as well as portability across multiple rendering systems (including pbrt).

See Ebert et al. (2003) and Apodaca and Gritz (2000) for techniques for writing procedural shaders; both of those have excellent discussions of issues related to antialiasing in procedural shaders.

The “Further Reading” section in Chapter 9 described approaches for anti-aliasing bump maps; a number of researchers have looked at the closely related issue of antialiasing surface reflection functions. Van Horn and Turk (2008) developed an approach to automatically generate MIP maps of reflection functions that represent the characteristics of shaders over finite areas in order to antialias them. Bruneton and Neyret (2012) surveyed the state of the art in this area, and Jarabo et al. (2014b) also considered perceptual issues related to filtering inputs to these functions. See also Heitz et al. (2014) for recent work on this topic.

Many creative methods for computing texture on surfaces have been developed. A sampling of our favorites includes reaction diffusion, which simulates growth processes based on a model of chemical interactions over surfaces and was simultaneously introduced by Turk (1991) and Witkin and Kass (1991); Sims’s (1991) genetic algorithm-based approach, which finds programs that generate interesting textures through random mutations from which users select their favorites; Fleischer et al.’s (1995) cellular texturing algorithms that generate geometrically accurate scales and spike features on surfaces; and Dorsey et al.’s (1996) flow simulations that model the effect of weathering on buildings and encode the results in image maps that stored the relative wetness, dirtiness, and so on, at points on the surfaces of structures. Porumbescu et al. (2005) developed *shell maps*, which make it possible to map geometric objects onto a surface in the manner of texture mapping.

A variety of *texture synthesis* algorithms have been developed in the past decade; these approaches take an example texture image and then synthesize larger texture maps that appear similar to the original texture while not being exactly the same. The survey article by Wei et al. (2009) describes work in this area through 2009 as well as the main approaches that have been developed so far. For more recent work in this area, see Kim et al. (2012), who developed an effective approach based on finding symmetries in textures, and Lefebvre et al. (2010), who attacked the specialized (but useful) problem of synthesizing textures for building facades.

EXERCISES

- ② 10.1 Many image file formats don’t store floating-point color values but instead use 8 bits for each color component, mapping the values to the range [0, 1]. (For example, the TGA format that is supported by `ReadImage()` is such a format.) For images originally stored in this format, the `ImageTexture` uses four times more memory than strictly necessary by using floats in `RGBSpectrum` objects to store these colors. Modify the image reading routines to directly return 8-bit values when an image is read from such a file.

Then modify the `ImageTexture` so that it keeps the data for such textures in an 8-bit representation, and modify the `MIPMap` so that it can filter data stored in this format. How much memory is saved for image texture-heavy scenes? How is pbrt’s performance affected? Can you explain the causes of any performance differences?

- ② 10.2 For scenes with many image textures where reading them all into memory simultaneously has a prohibitive memory cost, an effective approach can be

`ImageTexture` 619

`MIPMap` 625

`RGBSpectrum` 332

to allocate a fixed amount of memory for image maps (a *texture cache*), load textures into that memory on demand, and discard the image maps that haven't been accessed recently when the memory fills up (Peachey 1990). To enable good performance with small texture caches, image maps should be stored in a *tiled* format that makes it possible to load in small square regions of the texture independently of each other. Tiling techniques like these are used in graphics hardware to improve the performance of their texture memory caches (Hakura and Gupta 1997; Igehy et al. 1998, 1999). Implement a texture cache in `pbrt`. Write a conversion program that converts images in other formats to a tiled format. (You may want to investigate OpenEXR's tiled image support.) How small can you make the texture cache and still see good performance?

- ➊ 10.3 Read the papers by Manson and Schaefer (2013, 2014) on approximating high-quality filters with MIP maps and a small number of bilinear samples. Add an option to use their method for texture filtering in place of the EWA implementation currently in `pbrt`. Compare image quality for a number of scenes that use textures. How does running time compare? You may also find it useful to use a profiler to compare the amount of time running texture filtering code for each of the two approaches.
- ➋ 10.4 Improve the filtering algorithm used for resampling image maps to initialize the MIP map levels using the Lanczos filter instead of the box filter. How do the sphere test images in the file `scenes/sphere-ewa-vs-trilerp.pbrt` and Figure 10.10 change after your improvements?
- ➌ 10.5 It is possible to use MIP mapping with textures that have non-power-of-two resolutions—the details are explained by Guthe and Heckbert (2005). Implementing this approach can save a substantial amount of memory: in the worst case, the resampling that `pbrt`'s `MIPMap` implementation performs can increase memory requirements by a factor of four. (Consider a 513×513 texture that is resampled to be 1024×1024 .) Implement this approach in `pbrt`, and compare the amount of memory used to store texture data for a variety of texture-heavy scenes.
- ➍ 10.6 Some of the light transport algorithms in Chapters 14–16 require a large number of samples to be taken per pixel for good results. (Examples of such algorithms include path tracing as implemented by the `PathIntegrator`.) If hundreds or thousands of samples are taken in each pixel, then the computational expense of high-quality texture filtering isn't worthwhile; the high pixel sampling rate serves well to antialias texture functions with high frequencies. Modify the `MIPMap` implementation so that it optionally just returns a bilinearly interpolated value from the finest level of the pyramid, even if a filter footprint is provided. Compare rendering time and image quality with this approach when rendering an image using many samples per pixel and a scene that has image maps that would otherwise exhibit aliasing at lower pixel sampling rates.
- ➎ 10.7 An additional advantage of properly antialiased image map lookups is that they improve cache performance. Consider, for example, the situation of undersampling a high-resolution image map: nearby samples on the screen will access widely separated parts of the image map, such that there is low probability

`MIPMap` 625

`PathIntegrator` 875

that texels fetched from main memory for one texture lookup will already be in the cache for texture lookups at adjacent pixel samples. Modify `pbrt` so that it always does image texture lookups from the finest level of the MIPMap, being careful to ensure that the same number of texels are still being accessed. How does performance change? What do cache-profiling tools report about the overall change in effectiveness of the CPU cache?

- ② 10.8 Read Worley’s paper that describes a new noise function with substantially different visual characteristics than Perlin noise (Worley 1996). Implement this cellular noise function, and add Textures to `pbrt` that are based on it.
- ② 10.9 Implement one of the improved noise functions, such as the ones introduced by Cook and DeRose (2005), Goldberg et al. (2008), or Lagae et al. (2009). Compare image quality and rendering time for scenes that make substantial use of noise functions to the current implementation in `pbrt`.
- ② 10.10 The implementation of the `DotsTexture` texture in this chapter does not make any effort to avoid aliasing in the results that it computes. Modify this texture to do some form of antialiasing. The `Checkerboard2DTexture` offers a guide as to how this might be done, although this case is more complicated, both because the polka dots are not present in every grid cell and because they are irregularly positioned.

At the two extremes of a filter region that is within a single cell and a filter region that spans a large number of cells, the task is easier. If the filter is entirely within a single cell and is entirely inside or outside the polka dot in that cell (if present), then it is only necessary to evaluate one of the two subtextures as appropriate. If the filter is within a single cell but overlaps both the dot and the base texture, then it is possible to compute how much of the filter area is inside the dot and how much is outside and blend between the two. At the other extreme, if the filter area is extremely large, it is possible to blend between the two textures according to the overall average of how much area is covered by dots and how much is not. (Note that this approach potentially makes the same error as was made in the checkerboard, where the subtextures aren’t aware that part of their area is occluded by another texture. Ignore this issue for this exercise.)

Implement these approaches and then consider the intermediate cases, where the filter region spans a small number of cells. What approaches work well for antialiasing in this case?

- ② 10.11 Write a general-purpose Texture that stores a reference to another texture and supersamples that texture when the evaluation method is called, thus making it possible to apply supersampling to any Texture. Use your implementation to compare the effectiveness and quality of the built-in antialiasing done by various procedural textures. Also compare the run-time efficiency of texture supersampling versus increased pixel sampling.
- ③ 10.12 Modify `pbrt` to support a shading language to allow user-written programs to compute texture values. Unless you’re also interested in writing your own compiler, *OSL* (Gritz et al. 2010) is a good choice.

`Checkerboard2DTexture` 642

`DotsTexture` 653

`Texture` 614



CHAPTER ELEVEN

★ 11 VOLUME SCATTERING

So far, we have assumed that scenes are made up of collections of surfaces in a vacuum, which means that radiance is constant along rays between surfaces. However, there are many real-world situations where this assumption is inaccurate: fog and smoke attenuate and scatter light, and scattering from particles in the atmosphere makes the sky blue and sunsets red. This chapter introduces the mathematics to describe how light is affected as it passes through *participating media*—large numbers of very small particles distributed throughout a region of 3D space. Volume scattering models are based on the assumption that there are so many particles that scattering is best modeled as a probabilistic process, rather than directly accounting for individual interactions with particles. Simulating the effect of participating media makes it possible to render images with atmospheric haze, beams of light through clouds, light passing through cloudy water, and subsurface scattering, where light exits a solid object at a different place than where it entered.

This chapter first describes the basic physical processes that affect the radiance along rays passing through participating media. It then introduces the `Medium` base class, which provides interfaces for describing participating media in a region of space. `Medium` implementations return information about the scattering properties at points in their extent, including a phase function, which characterizes how light is scattered at a point in space. (It's the volumetric analog to the BSDF, which describes scattering at a point on a surface.) In order to determine the effect of participating media on the distribution of radiance in the scene, Integrators that handle volumetric effects are necessary; this is the topic of Chapter 15.

In highly scattering participating media, light can undergo many scattering events without any appreciable reduction in its energy. The cost of finding a light path in an Integrator is generally proportional to its length, and tracking paths with hundreds or thousands of scattering interactions quickly becomes impractical. In such cases, it is

preferable to aggregate the overall effect of the underlying scattering process in a function that relates scattering between points where light enters and leaves the medium. The chapter therefore concludes with the BSSRDF base class, which is an abstraction that makes it possible to implement this type of approach. BSSRDF implementations describe the internal scattering in a medium bounded by refractive surfaces.

11.1 VOLUME SCATTERING PROCESSES

There are three main processes that affect the distribution of radiance in an environment with participating media:

- *Absorption*: the reduction in radiance due to the conversion of light to another form of energy, such as heat
- *Emission*: radiance that is added to the environment from luminous particles
- *Scattering*: radiance heading in one direction that is scattered to other directions due to collisions with particles

The characteristics of all of these properties may be *homogeneous* or *inhomogeneous*. Homogeneous properties are constant throughout some region of space given spatial extent, while inhomogeneous properties vary throughout space. Figure 11.1 shows a

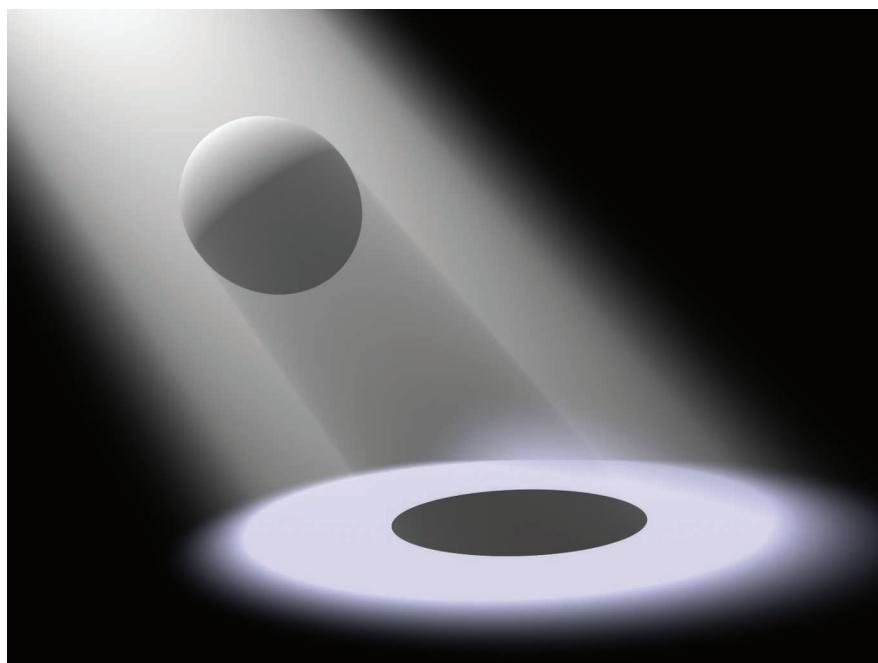


Figure 11.1: Spotlight through Fog. Light scattering from particles in the medium back toward the camera makes the spotlight's illumination visible even in pixels where there are no visible surfaces that reflect it. The sphere blocks light, casting a volumetric shadow in the region beneath it.

BSSRDF 692

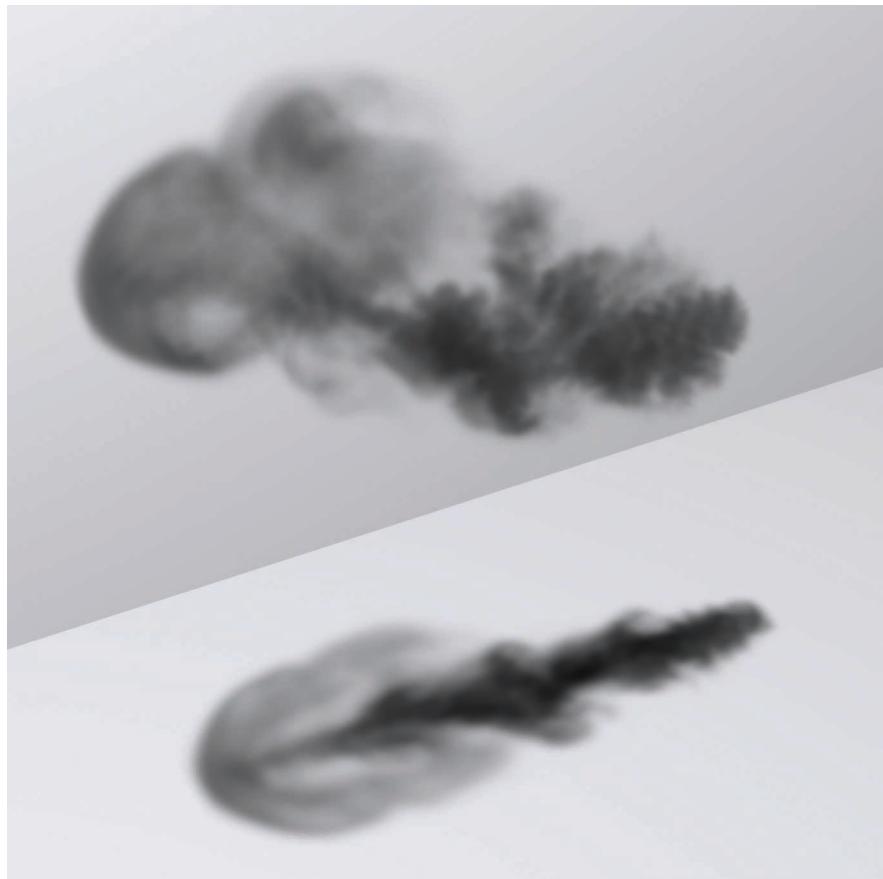


Figure 11.2: If a participating medium primarily absorbs light passing through it, it will have a dark and smoky appearance, as shown here. (*Smoke simulation data courtesy of Duc Nguyen and Ron Fedkiw.*)

simple example of volume scattering, where a spotlight shining through a participating medium illuminates particles in the medium and casts a volumetric shadow.

11.1.1 ABSORPTION

Consider thick black smoke from a fire: the smoke obscures the objects behind it because its particles absorb light traveling from the object to the viewer. The thicker the smoke, the more light is absorbed. Figure 11.2 shows this effect with a spatial distribution of absorption that was created with an accurate physical simulation of smoke formation. Note the shadow on the ground: the participating medium has also absorbed light between the light source to the ground plane, casting a shadow.

Absorption is described by the medium's *absorption cross section*, σ_a , which is the probability density that light is absorbed per unit distance traveled in the medium. In general, the absorption cross section may vary with both position p and direction ω , although it

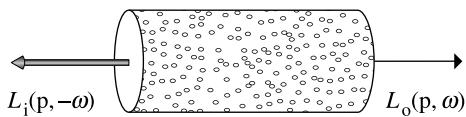


Figure 11.3: Absorption reduces the amount of radiance along a ray through a participating medium. Consider a ray carrying incident radiance at a point p from direction $-\omega$. If the ray passes through a differential cylinder filled with absorbing particles, the change in radiance due to absorption by those particles is $dL_o(p, \omega) = -\sigma_a(p, \omega)L_i(p, -\omega)dt$.

is normally just a function of position. It is usually also a spectrally varying quantity. The units of σ_a are reciprocal distance (m^{-1}). This means that σ_a can take on any positive value; it is not required to be between 0 and 1, for instance.

Figure 11.3 shows the effect of absorption along a very short segment of a ray. Some amount of radiance $L_i(p, -\omega)$ is arriving at point p , and we'd like to find the exitant radiance $L_o(p, \omega)$ after absorption in the differential volume. This change in radiance along the differential ray length dt is described by the differential equation

$$L_o(p, \omega) - L_i(p, -\omega) = dL_o(p, \omega) = -\sigma_a(p, \omega) L_i(p, -\omega) dt,$$

which says that the differential reduction in radiance along the beam is a linear function of its initial radiance.¹

This differential equation can be solved to give the integral equation describing the total fraction of light absorbed for a ray. If we assume that the ray travels a distance d in direction ω through the medium starting at point p , the remaining portion of the original radiance is given by

$$e^{-\int_0^d \sigma_a(p+t\omega, \omega) dt}.$$

11.1.2 EMISSION

While absorption reduces the amount of radiance along a ray as it passes through a medium, emission increases it, due to chemical, thermal, or nuclear processes that convert energy into visible light. Figure 11.4 shows emission in a differential volume, where we denote emitted radiance added to a ray per unit distance at a point p in direction ω by $L_e(p, \omega)$. Figure 11.5 shows the effect of emission with the smoke data set. In that figure the absorption coefficient is much lower than in Figure 11.2, giving a very different appearance.

The differential equation that gives the change in radiance due to emission is

$$dL_o(p, \omega) = L_e(p, \omega) dt.$$

¹ This is another instance of the linearity assumption in radiometry: the fraction of light absorbed doesn't vary based on the ray's radiance, but is always a fixed fraction.

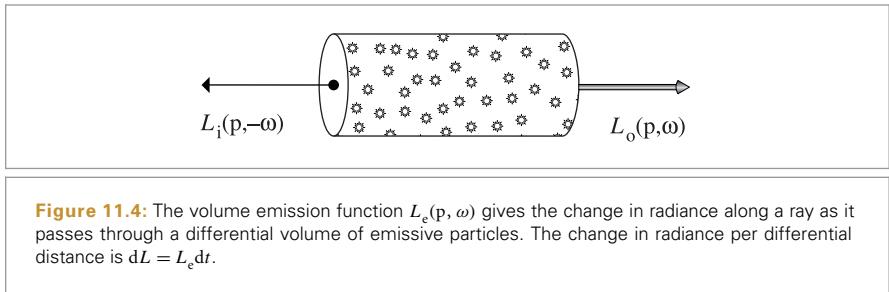


Figure 11.4: The volume emission function $L_e(p, \omega)$ gives the change in radiance along a ray as it passes through a differential volume of emissive particles. The change in radiance per differential distance is $dL = L_e dt$.



Figure 11.5: A Participating Medium Where the Dominant Volumetric Effect Is Emission.

Although the medium still absorbs light, still casting a shadow on the ground and obscuring the wall behind it, emission in the volume increases radiance along rays passing through it, making the cloud brighter than the wall behind it.

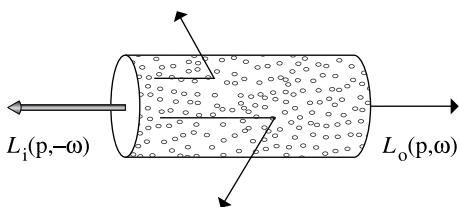


Figure 11.6: Like absorption, out-scattering also reduces the radiance along a ray. Light that hits particles may be scattered in another direction such that the radiance exiting the region in the original direction is reduced.

This equation incorporates the assumption that the emitted light L_e is not dependent on the incoming light L_i . This is always true under the linear optics assumptions that pbrt is based on.

11.1.3 OUT-SCATTERING AND ATTENUATION

The third basic light interaction in participating media is scattering. As a ray passes through a medium, it may collide with particles and be scattered in different directions. This has two effects on the total radiance that the beam carries. It reduces the radiance exiting a differential region of the beam because some of it is deflected to different directions. This effect is called *out-scattering* (Figure 11.6) and is the topic of this section. However, radiance from other rays may be scattered into the path of the current ray; this *in-scattering* process is the subject of the next section.

The probability of an out-scattering event occurring per unit distance is given by the scattering coefficient, σ_s . As with absorption, the reduction in radiance along a differential length dt due to out-scattering is given by

$$dL_o(p, \omega) = -\sigma_s(p, \omega) L_i(p, -\omega) dt.$$

The total reduction in radiance due to absorption and out-scattering is given by the sum $\sigma_a + \sigma_s$. This combined effect of absorption and out-scattering is called *attenuation* or *extinction*. For convenience the sum of these two coefficients is denoted by the attenuation coefficient σ_t :

$$\sigma_t(p, \omega) = \sigma_a(p, \omega) + \sigma_s(p, \omega).$$

Two values related to the attenuation coefficient will be useful in the following. The first is the *albedo*, which is defined as

$$\rho = \frac{\sigma_s}{\sigma_t}.$$

The albedo is always between 0 and 1; it describes the probability of scattering (versus absorption) at a scattering event. The second is the *mean free path*, $1/\sigma_t$, which gives the average distance that a ray travels in the medium before interacting with a particle.

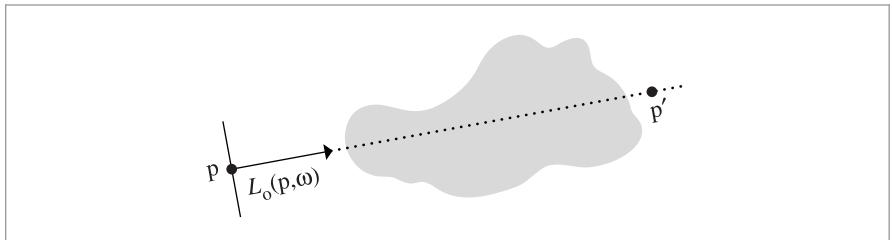


Figure 11.7: The beam transmittance $T_r(p \rightarrow p')$ gives the fraction of light transmitted from one point to another, accounting for absorption and out-scattering, but ignoring emission and in-scattering. Given exitant radiance at a point p in direction ω (e.g., reflected radiance from a surface), the radiance visible at another point p' along the ray is $T_r(p \rightarrow p')L_o(p, \omega)$.

Given the attenuation coefficient σ_t , the differential equation describing overall attenuation,

$$\frac{dL_o(p, \omega)}{dt} = -\sigma_t(p, \omega) L_i(p, -\omega),$$

can be solved to find the *beam transmittance*, which gives the fraction of radiance that is transmitted between two points:

$$T_r(p \rightarrow p') = e^{-\int_0^d \sigma_t(p+t\omega, \omega) dt} \quad [11.1]$$

where $d = \|p - p'\|$ is the distance between p and p' , ω is the normalized direction vector between them, and T_r denotes the beam transmittance between p and p' . Note that the transmittance is always between 0 and 1. Thus, if exitant radiance from a point p on a surface in a given direction ω is given by $L_o(p, \omega)$, after accounting for extinction, the incident radiance at another point p' in direction $-\omega$ is

$$T_r(p \rightarrow p') L_o(p, \omega).$$

This idea is illustrated in Figure 11.7.

Two useful properties of beam transmittance are that transmittance from a point to itself is 1, $T_r(p \rightarrow p) = 1$, and in a vacuum $\sigma_t = 0$ and so $T_r(p \rightarrow p') = 1$ for all p' . Furthermore, if the attenuation coefficient satisfies the directional symmetry $\sigma_t(\omega) = \sigma_t(-\omega)$ or does not vary with direction ω and only varies as function of position (this is generally the case), then the transmittance between two points is the same in both directions:

$$T_r(p \rightarrow p') = T_r(p' \rightarrow p).$$

This property follows directly from Equation (11.1).

Another important property, true in all media, is that transmittance is multiplicative along points on a ray:

$$T_r(p \rightarrow p'') = T_r(p \rightarrow p') T_r(p' \rightarrow p''), \quad [11.2]$$

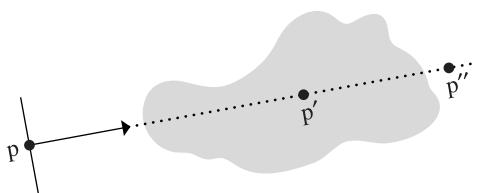


Figure 11.8: A useful property of beam transmittance is that it is multiplicative: the transmittance between points p and p'' on a ray like the one shown here is equal to the transmittance from p to p' times the transmittance from p' to p'' for all points p' between p and p'' .

for all points p' between p and p'' (Figure 11.8). This property is useful for volume scattering implementations, since it makes it possible to incrementally compute transmittance at multiple points along a ray: transmittance from the origin to a point $T_r(o \rightarrow p)$ can be computed by taking the product of transmittance to a previous point $T_r(o \rightarrow p')$ and the transmittance of the segment between the previous and the current point $T_r(p' \rightarrow p)$.

The negated exponent in the definition of T_r in Equation (11.1) is called the *optical thickness* between the two points. It is denoted by the symbol τ :

$$\tau(p \rightarrow p') = \int_0^d \sigma_t(p + t\omega, -\omega) dt.$$

In a homogeneous medium, σ_t is a constant, so the integral that defines τ is trivially evaluated, giving *Beer's law*:

$$T_r(p \rightarrow p') = e^{-\sigma_t d}. \quad [11.3]$$

11.1.4 IN-SCATTERING

While out-scattering reduces radiance along a ray due to scattering in different directions, *in-scattering* accounts for increased radiance due to scattering from other directions (Figure 11.9). Figure 11.10 shows the effect of in-scattering with the smoke data set. Note that the smoke appears much thicker than when absorption or emission was the dominant volumetric effect.

Assuming that the separation between particles is at least a few times the lengths of their radii, it is possible to ignore inter-particle interactions when describing scattering at a particular location. Under this assumption, the *phase function* $p(\omega, \omega')$ describes the angular distribution of scattered radiation at a point; it is the volumetric analog to the BSDF. The BSDF analogy is not exact, however. For example, phase functions have a normalization constraint: for all ω , the condition

$$\int_{S^2} p(\omega, \omega') d\omega' = 1 \quad [11.4]$$

must hold. This constraint means that phase functions actually define probability distributions for scattering in a particular direction.

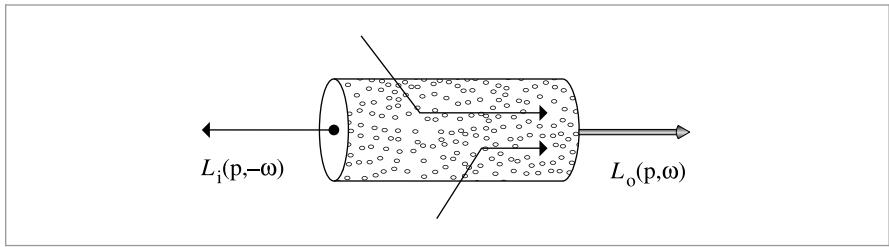


Figure 11.9: In-scattering accounts for the increase in radiance along a ray due to scattering of light from other directions. Radiance from outside the differential volume is scattered along the direction of the ray and added to the incoming radiance.



Figure 11.10: In-Scattering with the Smoke Data Set. Note the substantially different appearance compared to the other two smoke images.

The total added radiance per unit distance due to in-scattering is given by the *source term* L_s :

$$dL_o(p, \omega) = L_s(p, \omega) dt.$$

It accounts for both volume emission and in-scattering:

$$L_s(p, \omega) = L_e(p, \omega) + \sigma_s(p, \omega) \int_{S^2} p(p, \omega_i, \omega) L_i(p, \omega_i) d\omega_i.$$

The in-scattering portion of the source term is the product of the scattering probability per unit distance, σ_s , and the amount of added radiance at a point, which is given by the spherical integral of the product of incident radiance and the phase function. Note that the source term is very similar to the scattering equation, Equation (5.9); the main difference is that there is no cosine term since the phase function operates on radiance rather than differential irradiance.

11.2 PHASE FUNCTIONS

Just as there is a wide variety of BSDF models that describe scattering from surfaces, many phase functions have also been developed. These range from parameterized models (which can be used to fit a function with a small number of parameters to measured data) to analytic models that are based on deriving the scattered radiance distribution that results from particles with known shape and material (e.g., spherical water droplets).

In most naturally occurring media, the phase function is a 1D function of the angle θ between the two directions ω_o and ω_i ; these phase functions are often written as $p(\cos \theta)$. Media with this type of phase function are called *isotropic* because their response to incident illumination is (locally) invariant under rotations. In addition to being normalized, an important property of naturally occurring phase functions is that they are *reciprocal*: the two directions can be interchanged and the phase function's value remains unchanged. Note that isotropic phase functions are trivially reciprocal because $\cos(-\theta) = \cos(\theta)$.

In *anisotropic* media that consist of particles arranged in a coherent structure, the phase function can be a 4D function of the two directions, which satisfies a more involved kind of reciprocity relation. Examples of this are crystals or media made of coherently oriented fibers; the “Further Reading” discusses these types of media further.

In a slightly confusing overloading of terminology, phase functions themselves can be isotropic or anisotropic as well. Thus, we might have an anisotropic phase function in an isotropic medium. An isotropic phase function describes equal scattering in all directions and is thus independent of either of the two directions. Because phase functions are normalized, there is only one such function:

$$p(\omega_o, \omega_i) = \frac{1}{4\pi}.$$

The `PhaseFunction` abstract base class defines the interface for phase functions in `pbrt`.

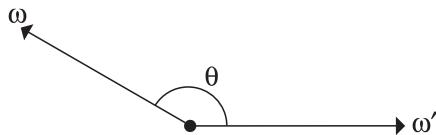


Figure 11.11: Phase functions in pbrt are implemented with the convention that both the incident direction and the outgoing direction point away from the point where scattering happens. This is the same convention that is used for BSDFs in pbrt but is different from the convention in the scattering literature, where the incident direction generally points toward the scattering point. The angle between the two directions is denoted by θ .

```
(Media Declarations) ≡
    class PhaseFunction {
        public:
            (PhaseFunction Interface 681)
    };
```

The `p()` method returns the value of the phase function for the given pair of directions. As with BSDFs, pbrt uses the convention that the two directions both point away from the point where scattering occurs; this is a different convention from what is usually used in the scattering literature (Figure 11.11).

```
(PhaseFunction Interface) ≡
    virtual Float p(const Vector3f &wo, const Vector3f &wi) const = 0;
```

681

A widely used phase function was developed by Henyey and Greenstein (1941). This phase function was specifically designed to be easy to fit to measured scattering data. A single parameter g (called the *asymmetry parameter*) controls the distribution of scattered light.²

$$p_{\text{HG}}(\cos \theta) = \frac{1}{4\pi} \frac{1 - g^2}{(1 + g^2 + 2g(\cos \theta))^{3/2}}.$$

The `PhaseHG()` function implements this computation.

```
(Media Inline Functions) ≡
    inline Float PhaseHG(Float cosTheta, Float g) {
        Float denom = 1 + g * g + 2 * g * cosTheta;
        return Inv4Pi * (1 - g * g) / (denom * std::sqrt(denom));
    }
```

Float 1062

Inv4Pi 1063

Vector3f 60

² Note that the sign of the $2g(\cos \theta)$ term in the denominator is the opposite of the sign used in the scattering literature. This difference is due to our use of the same direction convention for BSDFs and phase functions.

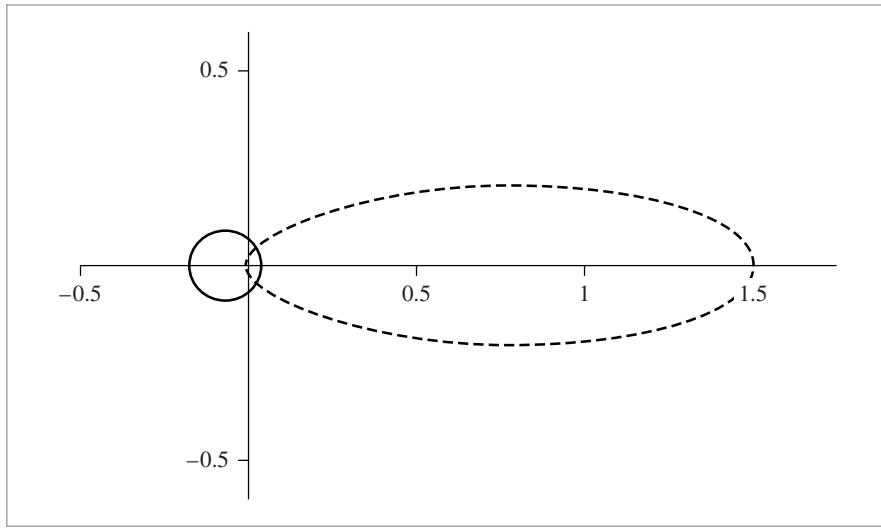


Figure 11.12: Plots of the Henyey–Greenstein Phase Function for Asymmetry g Parameters –0.35 and 0.67. Negative g values (solid line) describe phase functions that primarily scatter light back in the incident direction, and positive g values (dashed line) describe phase functions that primarily scatter light forward in the direction it was already traveling.

Figure 11.12 shows plots of the Henyey–Greenstein phase function with varying asymmetry parameters. The value of g for this model must be in the range $(-1, 1)$. Negative values of g correspond to *back-scattering*, where light is mostly scattered back toward the incident direction, and positive values correspond to forward-scattering. The greater the magnitude of g , the more scattering occurs close to the ω or $-\omega$ directions (for back-scattering and forward-scattering, respectively). See Figure 11.13 to compare the visual effect of forward- and back-scattering.

`HenyeyGreenstein` provides a `PhaseFunction` implementation of the Henyey–Greenstein model.

```
(HenyeyGreenstein Declarations) ≡
class HenyeyGreenstein : public PhaseFunction {
public:
    (HenyeyGreenstein Public Methods 682)
private:
    const Float g;
};
```

```
(HenyeyGreenstein Public Methods) ≡
HenyeyGreenstein(Float g) : g(g) { }
```

```
(HenyeyGreenstein Method Definitions) ≡
Float HenyeyGreenstein::p(const Vector3f &wo, const Vector3f &wi) const {
    return PhaseHG(Dot(wo, wi), g);
}
```

Dot() 63 Float 1062 HenyeyGreenstein 682 HenyeyGreenstein::g 682 PhaseFunction 681 PhaseHG() 681 Vector3f 60
--



Figure 11.13: Objects filled with participating media rendered with (left) strong backward scattering ($g = -0.7$) and (right) strong forward scattering ($g = 0.7$). Because the light source is behind the object with respect to the viewer, forward scattering leads to more light reaching the camera in this case.

The asymmetry parameter g in the Henyey–Greenstein model has a precise meaning. It is the average value of the product of the phase function being approximated and the cosine of the angle between ω' and ω . Given an arbitrary phase function p , the value of g can be computed as³

$$g = \int_{\mathbb{S}^2} p(-\omega, \omega')(\omega \cdot \omega') \, d\omega' = 2\pi \int_0^\pi p(-\cos \theta) \cos \theta \sin \theta \, d\theta. \quad [11.5]$$

Thus, an isotropic phase function gives $g = 0$, as expected.

Any number of phase functions can satisfy this equation; the g value alone is not enough to uniquely describe a scattering distribution. Nevertheless, the convenience of being able to easily convert a complex scattering distribution into a simple parameterized model is often more important than this potential loss in accuracy.

More complex phase functions that aren't described well with a single asymmetry parameter can often be modeled by a weighted sum of phase functions like Henyey–Greenstein, each with different parameter values:

$$p(\omega, \omega') = \sum_{i=1}^n w_i p_i(\omega \rightarrow \omega'),$$

where the weights w_i sum to one to maintain normalization. This generalization isn't provided in pbrt but would be easy to add.

³ Once more, there is a sign difference compared to the radiative transfer literature: the first argument to p is negated due to our use of the same direction convention for BSDFs and phase functions.

11.3 MEDIA

Implementations of the `Medium` base class provide various representations of volumetric scattering properties in a region of space. In a complex scene, there may be multiple `Medium` instances, each representing a different scattering effect. For example, an outdoor lake scene might have one `Medium` to model atmospheric scattering, another to model mist rising from the lake, and a third to model particles suspended in the water of the lake.

```
(Medium Declarations) ≡
class Medium {
public:
    (Medium Interface 684)
};
```

A key operation that `Medium` implementations must perform is to compute the beam transmittance, Equation (11.1), along a given ray passed to its `Tr()` method. Specifically, the method should return an estimate of the transmittance on the interval between the ray origin and the point at a distance of `Ray::tMax` from the origin.

Medium-aware Integrators using this interface are responsible for accounting for interactions with surfaces in the scene as well as the spatial extent of the `Medium`; hence we will assume that the ray passed to the `Tr()` method is both unoccluded and fully contained within the current `Medium`. Some implementations of this method use Monte Carlo integration to compute the transmittance; a `Sampler` is provided for this case. (See Section 15.2.)

```
(Medium Interface) ≡
virtual Spectrum Tr(const Ray &ray, Sampler &sampler) const = 0;
```

684

The spatial distribution and extent of media in a scene is defined by associating `Medium` instances with the camera, lights, and primitives in the scene. For example, Cameras store a `Medium` pointer that gives the medium for rays leaving the camera and similarly for Lights.

In pbrt, the boundary between two different types of scattering media is always represented by the surface of a `GeometricPrimitive`. Rather than storing a single `Medium` pointer like lights and cameras, `GeometricPrimitives` hold a `MediumInterface`, which in turn holds pointers to one `Medium` for the interior of the primitive and one for the exterior. For all of these cases, a `nullptr` can be used to indicate a vacuum (where no volumetric scattering occurs.)

```
(MediumInterface Declarations) ≡
struct MediumInterface {
    (MediumInterface Public Methods 685)
    const Medium *inside, *outside;
};
```

This approach to specifying the extent of participating media does allow the user to specify impossible or inconsistent configurations. For example, a primitive could be specified as having one medium outside of it, and the camera could be specified as being

`Camera` 356
`GeometricPrimitive` 250
`Integrator` 25
`Light` 714
`Medium` 684
`Ray` 73
`Ray::tMax` 73
`Sampler` 421
`Spectrum` 315

in a different medium without a `MediumInterface` between the camera and the surface of the primitive. In this case, a ray leaving the primitive toward the camera would be treated as being in a different medium from a ray leaving the camera toward the primitive. In turn, light transport algorithms would be unable to compute consistent results. For `pbrt`'s purposes, we think it's reasonable to expect that the user will be able to specify a consistent configuration of media in the scene and that the added complexity of code to check this isn't worthwhile.

A `MediumInterface` can be initialized with either one or two `Medium` pointers. If only one is provided, then it represents an interface with the same medium on both sides.

(MediumInterface Public Methods) ≡

```
MediumInterface(const Medium *medium)
    : inside(medium), outside(medium) { }
MediumInterface(const Medium *inside, const Medium *outside)
    : inside(inside), outside(outside) { }
```

684

The function `MediumInterface::IsMediumTransition()` checks whether a particular `MediumInterface` instance marks a transition between two distinct media.

(MediumInterface Public Methods) +≡

```
bool IsMediumTransition() const { return inside != outside; }
```

684

We can now provide a missing piece in the implementation of the `GeometricPrimitive::Intersect()` method. The code in this fragment is executed whenever an intersection with a geometric primitive has been found; its job is to set the medium interface at the intersection point.

Instead of simply copying the value of the `GeometricPrimitive::mediumInterface` field, we will follow a slightly different approach and only use this information when this `MediumInterface` specifies a proper transition between participating media. Otherwise, the `Ray::medium` field takes precedence.

Setting the `SurfaceInteraction`'s `mediumInterface` field in this way greatly simplifies the specification of scenes containing media: in particular, it is not necessary to tag every scene surface that is in contact with a medium. Instead, only non-opaque surfaces that have different media on each side require an explicit medium reference in their `GeometricPrimitive::mediumInterface` field. In the simplest case where a scene containing opaque objects is filled with a participating medium (e.g., haze), it is enough to tag the camera and light sources.

```
GeometricPrimitive::
    Intersect()
    251

GeometricPrimitive::
    mediumInterface
    250

GlassMaterial 584

Interaction::mediumInterface
    116

Material 577

Medium 684

MediumInterface 684

MediumInterface::
    IsMediumTransition()
    685

Ray::medium 74

SurfaceInteraction 116
```

(Initialize SurfaceInteraction::mediumInterface after Shape intersection) ≡

```
if (mediumInterface.IsMediumTransition())
    isect->mediumInterface = mediumInterface;
else
    isect->mediumInterface = MediumInterface(r.medium);
```

251

Primitives associated with shapes that represent medium boundaries generally have a `Material` associated with them. For example, the surface of a lake might use an instance of `GlassMaterial` to describe scattering at the lake surface, which also acts as the boundary between the rising mist's `Medium` and the lake water's `Medium`. However, sometimes



Figure 11.14: Scattering Media inside the Dragon. Both dragon models have the same homogeneous scattering media inside of them. On the left, the dragon's surface has a glass material. On the right, the dragon's `Material *` is `nullptr`, which indicates that the surface should be ignored by rays and is only used to delineate a participating medium's extent.

we only need the shape for the boundary surface it provides to delimit a participating medium boundary and we don't want to see the surface itself. For example, the medium representing a cloud might be bounded by a box made of triangles where the triangles are only there to delimit the cloud's extent and shouldn't otherwise affect light passing through them.

While such a surface that disappears and doesn't affect ray paths could be perfectly accurately described by a BTDF that represented perfect specular transmission with the same index of refraction on both sides, dealing with such surfaces places extra burden on the Integrators (not all of which handle this type of specular light transport well). Therefore, pbrt allows such surfaces to have a `Material *` that is `nullptr`, indicating that they do not affect light passing through them; in turn, `SurfaceInteraction::bsdf` will also be `nullptr`, and the light transport routines don't worry about light scattering from such surfaces and only account for changes in the current medium at them. Figure 11.14 has two instances of the dragon model filled with scattering media; one has a scattering surface at the boundary and the other does not.

Given these conventions for how Medium implementations are associated with rays passing through regions of space, we will implement a `Scene::IntersectTr()` method, which is a generalization of `Scene::Intersect()` that returns the first intersection with a light-scattering surface along the given ray as well as the beam transmittance up to that point. (If no intersection is found, this method returns `false` and doesn't initialize the provided `SurfaceInteraction`.)

`Scene::Intersect()` 24
`SurfaceInteraction` 116
`SurfaceInteraction::bsdf` 250

```

<Scene Method Definitions> +≡
    bool Scene::IntersectTr(Ray ray, Sampler &sampler,
        SurfaceInteraction *isect, Spectrum *Tr) const {
        *Tr = Spectrum(1.f);
        while (true) {
            bool hitSurface = Intersect(ray, isect);
            <Accumulate beam transmittance for ray segment 687>
            <Initialize next ray segment or terminate transmittance computation 687>
        }
    }
}

```

Each time through the loop, the transmittance along the ray is accumulated into the overall beam transmittance `*Tr`. Recall that `Scene::Intersect()` will have updated the ray's `tMax` member variable to the intersection point if it did intersect a surface. The `Tr()` implementation will use this value to find the segment over which to compute transmittance.

```

<Accumulate beam transmittance for ray segment> ≡
if (ray.medium)
    *Tr *= ray.medium->Tr(ray, sampler);

```

687

The loop ends when no intersection is found or when a scattering surface is intersected. If an optically inactive surface with its `bsdf` equal to `nullptr` is intersected, a new ray is spawned in the same direction from the intersection point, though potentially in a different medium, based on the intersection's `MediumInterface` field.⁴

```

<Initialize next ray segment or terminate transmittance computation> ≡
if (!hitSurface)
    return false;
if (isect->primitive->GetMaterial() != nullptr)
    return true;
ray = isect->SpawnRay(ray.d);

```

687

11.3.1 MEDIUM INTERACTIONS

Section 2.10 introduced the general `Interaction` class as well as the `SurfaceInteraction` specialization to represent interactions at surfaces. Now that we have some machinery for describing scattering in volumes, it's worth generalizing these representations. First, we'll add two more `Interaction` constructors for interactions at points in scattering media.

```

<Interaction Public Methods> +≡
Interaction(const Point3f &p, const Vector3f &wo, Float time,
            const MediumInterface &mediumInterface)
: p(p), time(time), wo(wo), mediumInterface(mediumInterface) { }

```

115

```

Float 1062
Interaction 115
Interaction::mediumInterface 116
Interaction::p 115
Interaction::SpawnRay() 232
Interaction::time 115
Interaction::wo 115
Medium::Tr() 684
MediumInterface 684
Point3f 68
Primitive::GetMaterial() 249
Ray 73
Ray::medium 74
Sampler 421
Scene::Intersect() 24
Spectrum 315
SurfaceInteraction 116
SurfaceInteraction::primitive 249
Vector3f 60

```

⁴ If the current medium does change, it should have the same index of refraction as the previous one; otherwise, there should be a `Material` with a `BTDF` that describes the effect of refraction on the ray's direction. The implementation here doesn't verify that the two indices of refraction match, however.

(Interaction Public Methods) +≡ 115

```
Interaction(const Point3f &p, Float time,
            const MediumInterface &mediumInterface)
: p(p), time(time), mediumInterface(mediumInterface) { }
```

(Interaction Public Methods) +≡ 115

```
bool IsMediumInteraction() const { return !IsSurfaceInteraction(); }
```

For surface interactions where `Interaction::n` has been set, the `Medium *` for a ray leaving the surface in the direction `w` is returned by the `GetMedium()` method.

(Interaction Public Methods) +≡ 115

```
const Medium *GetMedium(const Vector3f &w) const {
    return Dot(w, n) > 0 ? mediumInterface.outside :
                           mediumInterface.inside;
}
```

For interactions that are known to be inside participating media, another variant of `GetMedium()` that doesn't take the unnecessary outgoing direction vector returns the `Medium *`.

(Interaction Public Methods) +≡ 115

```
const Medium *GetMedium() const {
    Assert(mediumInterface.inside == mediumInterface.outside);
    return mediumInterface.inside;
}
```

Just as the `SurfaceInteraction` class represents an interaction obtained by intersecting a ray against the scene geometry, `MediumInteraction` represents an interaction at a point in a scattering medium that is obtained using a similar kind of operation.

(Interaction Declarations) +≡

```
class MediumInteraction : public Interaction {
public:
    (MediumInteraction Public Methods 688)
    (MediumInteraction Public Data 688)
};
```

(MediumInteraction Public Methods) ≡ 688

```
MediumInteraction(const Point3f &p, const Vector3f &wo, Float time,
                  const Medium *medium, const PhaseFunction *phase)
: Interaction(p, wo, time, medium), phase(phase) { }
```

Dot() 63
 Float 1062
 Interaction 115
 Interaction::
 IsSurfaceInteraction()
 116
 Interaction::mediumInterface
 116
 Interaction::n 116
 Interaction::p 115
 Interaction::time 115
 Medium 684
 MediumInterface 684
 MediumInterface::inside 684
 MediumInterface::outside 684
 PhaseFunction 681
 Point3f 68
 SurfaceInteraction 116
 Vector3f 60

`MediumInteraction` adds a new `PhaseFunction` member variable to store the phase function associated with its position.

(MediumInteraction Public Data) ≡ 688

```
const PhaseFunction *phase;
```

11.3.2 HOMOGENEOUS MEDIUM

The `HomogeneousMedium` is the simplest possible medium; it represents a region of space with constant σ_a and σ_s values throughout its extent. It uses the Henyey–Greenstein

phase function to represent scattering in the medium, also with a constant g value. This medium was used for the images in Figure 11.13 and 11.14.

```
<HomogeneousMedium Declarations> ≡
    class HomogeneousMedium : public Medium {
        public:
            <HomogeneousMedium Public Methods 689>
        private:
            <HomogeneousMedium Private Data 689>
    };

<HomogeneousMedium Public Methods> ≡
    HomogeneousMedium(const Spectrum &sigma_a, const Spectrum &sigma_s,
                      Float g)
    : sigma_a(sigma_a), sigma_s(sigma_s), sigma_t(sigma_s + sigma_a),
      g(g) { }

<HomogeneousMedium Private Data> ≡
    const Spectrum sigma_a, sigma_s, sigma_t;
    const Float g;
```

689

689

Because σ_t is constant throughout the medium, Beer's law, Equation (11.3), can be used to compute transmittance along the ray. However, implementation of the `Tr()` method is complicated by some subtleties of floating-point arithmetic. As discussed in Section 3.9.1, IEEE floating point provides a representation for infinity; in pbrt, this value, `Infinity`, is used to initialize `Ray::tMax` for rays leaving cameras and lights, which is useful for ray–intersection tests in that it ensures that any actual intersection, even if far along the ray, is detected as an intersection for a ray that hasn't intersected anything yet.

However, the use of `Infinity` for `Ray::tMax` creates a small problem when applying Beer's law. In principle, we just need to compute the parametric t range that the ray spans, multiply by the ray direction's length, and then multiply by σ_t :

```
Float d = ray.tMax * ray.Length();
Spectrum tau = sigma_t * d;
return Exp(-tau);
```

Float 1062
 HomogeneousMedium 689
 HomogeneousMedium::sigma_t 689
 Infinity 210
 MaxFloat 210
 Medium 684
 Ray 73
 Ray::tMax 73
 Sampler 421
 Spectrum 315
 Spectrum::Exp() 317

The problem is that multiplying `Infinity` by zero results in the floating-point “not a number” (NaN) value, which propagates throughout all computations that use it. For a ray that passes infinitely far through a medium with zero absorption for a given spectral channel, the above code would attempt to perform the multiplication $0 * \text{Infinity}$ and would produce a NaN value rather than the expected transmittance of zero. The implementation here resolves this issue by clamping the ray segment length to the largest representable non-infinite floating-point value.

```
<HomogeneousMedium Method Definitions> ≡
    Spectrum HomogeneousMedium::Tr(const Ray &ray, Sampler &sampler) const {
        return Exp(-sigma_t * std::min(ray.tMax * ray.d.Length(), MaxFloat));
    }
```

11.3.3 3D GRIDS

The `GridDensityMedium` class stores medium densities at a regular 3D grid of positions, similar to the way that the `ImageTexture` represents images with a 2D grid of samples. These samples are interpolated to compute the density at positions between the sample points. The implementation of the `GridDensityMedium` is in `media/grid.h` and `media/grid.cpp`.

```
(GridDensityMedium Declarations) ≡
class GridDensityMedium : public Medium {
public:
    (GridDensityMedium Public Methods 690)
private:
    (GridDensityMedium Private Data 690)
};
```

The constructor takes a 3D array of user-supplied density values, thus allowing a variety of sources of data (physical simulation, CT scan, etc.). The smoke data set rendered in Figures 11.2, 11.5, and 11.10 is represented with a `GridDensityMedium`. The caller also supplies baseline values of σ_a , σ_s , and g to the constructor, which does the usual initialization of the basic scattering properties and makes a local copy of the density values.

```
(GridDensityMedium Public Methods) ≡ 690
GridDensityMedium(const Spectrum &sigma_a, const Spectrum &sigma_s,
                  Float g, int nx, int ny, int nz, const Transform &mediumToWorld,
                  const Float *d)
: sigma_a(sigma_a), sigma_s(sigma_s), g(g), nx(nx), ny(ny), nz(nz),
  WorldToMedium(Inverse(mediumToWorld)),
  density(new Float[nx * ny * nz]) {
    memcpy((Float *)density.get(), d, sizeof(Float) * nx * ny * nz);
    (Precompute values for Monte Carlo sampling of GridDensityMedium 896)
}
```

```
(GridDensityMedium Private Data) ≡ 690
const Spectrum sigma_a, sigma_s;
const Float g;
const int nx, ny, nz;
const Transform WorldToMedium;
std::unique_ptr<Float[]> density;

Float 1062
GridDensityMedium 690
GridDensityMedium::Tr() 898
ImageTexture 619
Inverse() 1081
Medium 684
Spectrum 315
Transform 83
```

The `Density()` method of `GridDensityMedium` is called by `GridDensityMedium::Tr()`; it uses the provided samples to reconstruct the volume density function at the given point, which will already have been transformed into local coordinates using `WorldToMedium`. In turn, σ_a and σ_s will be scaled by the interpolated density at the point.

```
<GridDensityMedium Method Definitions> ≡
    Float GridDensityMedium::Density(const Point3f &p) const {
        <Compute voxel coordinates and offsets for p 691>
        <Trilinearly interpolate density values to compute local density 691>
    }
```

The grid samples are assumed to be over a canonical $[0, 1]^3$ domain. (The `WorldToMedium` transformation should be used to place the `GridDensityMedium` in the scene.) To interpolate the samples around a point, the `Density()` method first computes the coordinates of the point with respect to the sample coordinates and the distances from the point to the samples below it (along the lines of what was done in the `Film` and `MIPMap`—see also Section 7.1.7).

```
<Compute voxel coordinates and offsets for p> ≡ 691
    Point3f pSamples(p.x * nx - .5f, p.y * ny - .5f, p.z * nz - .5f);
    Point3i pi = (Point3i)Floor(pSamples);
    Vector3f d = pSamples - (Point3f)pi;
```

The distances `d` can be used directly in a series of invocations of `Lerp()` to trilinearly interpolate the density at the sample point:

```
<Trilinearly interpolate density values to compute local density> ≡ 691
    Float d00 = Lerp(d.x, D(pi), D(pi+Vector3i(1,0,0)));
    Float d10 = Lerp(d.x, D(pi+Vector3i(0,1,0)), D(pi+Vector3i(1,1,0)));
    Float d01 = Lerp(d.x, D(pi+Vector3i(0,0,1)), D(pi+Vector3i(1,0,1)));
    Float d11 = Lerp(d.x, D(pi+Vector3i(0,1,1)), D(pi+Vector3i(1,1,1)));
    Float d0 = Lerp(d.y, d00, d10);
    Float d1 = Lerp(d.y, d01, d11);
    return Lerp(d.z, d0, d1);
```

`Bounds3::InsideExclusive() 79`

`Bounds3i 76`

`Film 484`

`Float 1062`

`GridDensityMedium 690`

`GridDensityMedium::D() 691`

`GridDensityMedium::nx 690`

`GridDensityMedium::ny 690`

`GridDensityMedium::nz 690`

`Lerp() 1079`

`MIPMap 625`

`Point3i::Floor() 71`

`Point3f 68`

`Point3i 68`

`Vector3f 60`

`Vector3i 60`

The `D()` utility method returns the density at the given integer sample position. Its only tasks are to handle out-of-bounds sample positions and to compute the appropriate array offset for the given sample. Unlike MIPMaps, where a variety of behavior is useful in the case of out-of-bounds coordinates, here it's reasonable to always return a zero density for them: the density is defined over a particular domain, and it's reasonable that points outside of it should have zero density.

```
<GridDensityMedium Public Methods> +≡ 690
    Float D(const Point3i &p) const {
        Bounds3i sampleBounds(Point3i(0, 0, 0), Point3i(nx, ny, nz));
        if (!InsideExclusive(p, sampleBounds))
            return 0;
        return density[(p.z * ny + p.y) * nx + p.x];
    }
```

11.4 THE BSSRDF

The bidirectional scattering-surface reflectance distribution function (BSSRDF) was introduced in Section 5.6.2; it gives exitant radiance at a point on a surface p_o given incident differential irradiance at another point p_i : $S(p_o, \omega_o, p_i, \omega_i)$. Accurately rendering translucent surfaces with subsurface scattering requires integrating over both area—points on the surface of the object being rendered—and incident direction, evaluating the BSSRDF and computing reflection with the subsurface scattering equation

$$L_o(p_o, \omega_o) = \int_A \int_{\mathcal{H}^2(n)} S(p_o, \omega_o, p_i, \omega_i) L_i(p_i, \omega_i) |\cos \theta_i| d\omega_i dA.$$

Subsurface light transport is described by the volumetric scattering processes introduced in Sections 11.1 and 11.2 as well as the volume light transport equation that will be introduced in Section 15.1. The BSSRDF S is a summarized representation modeling the outcome of these scattering processes between a given pair of points and directions on the boundary.

A variety of BSSRDF models have been developed to model subsurface reflection; they generally include some simplifications to the underlying scattering processes to make them tractable. One such model will be introduced in Section 15.5. Here, we will begin by specifying a fairly abstract interface analogous to BSDF. All of the code related to BSSRDFs is in the files `core/bssrdf.h` and `core/bssrdf.cpp`.

(BSSRDF Declarations) ≡

```
class BSSRDF {
public:
    (BSSRDF Public Methods 692)
    (BSSRDF Interface 693)
protected:
    (BSSRDF Protected Data 692)
};
```

BSSRDF implementations must pass the current (outgoing) surface interaction as well as the index of refraction of the scattering medium to the base class constructor. There is thus an implicit assumption that the index of refraction is constant throughout the medium, which is a widely used assumption in BSSRDF models.

(BSSRDF Public Methods) ≡

692

```
BSSRDF(const SurfaceInteraction &po, Float eta)
: po(po), eta(eta) {}
```

(BSSRDF Protected Data) ≡

692

```
const SurfaceInteraction &po;
Float eta;
```

The key method that BSSRDF implementations must provide is one that evaluates the eight-dimensional distribution function $S()$, which quantifies the ratio of differential radiance at point p_o in direction ω_o to the incident differential flux at p_i from direction ω_i (Section 5.6.2). Since the p_o and ω_o arguments are already available via the `BSSRDF::po` and `Interaction::wo` fields, they aren't included in the method signature.

BSDF 572

BSSRDF::po 692

Float 1062

Interaction::wo 115

SurfaceInteraction 116

⟨BSSRDF Interface⟩ ≡

692

```
virtual Spectrum S(const SurfaceInteraction &pi, const Vector3f &wi) = 0;
```

Like the BSDF, the BSSRDF interface also defines functions to sample the distribution and to evaluate the probability density of the implemented sampling scheme. The specifics of this part of the interface are discussed in Section 15.4.

During the shading process, the current Material’s ComputeScatteringFunctions() method initializes the SurfaceInteraction::bssrdf member variable with an appropriate BSSRDF if the material exhibits subsurface scattering. (Section 11.4.3 will define two materials for subsurface scattering.)

11.4.1 SEPARABLE BSSRDFS

One issue with the BSSRDF interface as defined above is its extreme generality. Finding solutions to the subsurface light transport even in simple planar or spherical geometries is already a fairly challenging problem, and the fact that BSSRDF implementations can be attached to arbitrary and considerably more complex Shapes leads to an impractically difficult context. To retain the ability to support general Shapes, we’ll introduce a simpler BSSRDF representation in SeparableBSSRDF.

⟨BSSRDF Declarations⟩ +≡

```
class SeparableBSSRDF : public BSSRDF {
public:
    ⟨SeparableBSSRDF Public Methods 693⟩
    ⟨SeparableBSSRDF Interface 695⟩
private:
    ⟨SeparableBSSRDF Private Data 693⟩
};
```

BSDF 572
 BSSRDF 692
 Cross() 65
 Float 1062
 Material 577
 Normal3f 71
 SeparableBSSRDF 693
 SeparableBSSRDF::material 693
 SeparableBSSRDF::mode 693
 SeparableBSSRDF::ns 693
 SeparableBSSRDF::ss 693
 Shape 123
 Spectrum 315
 SurfaceInteraction 116
 SurfaceInteraction::bssrdf 250
 SurfaceInteraction::shading::dpdu 118
 SurfaceInteraction::shading::n 118
 TransportMode 960
 Vector3f::Normalize() 66
 Vector3f 60

The constructor of SeparableBSSRDF initializes a local coordinate frame defined by ss, ts, and ns, records the current light transport mode mode, and keeps a pointer to the underlying Material. The need for these values will be clarified in Section 15.4.

⟨SeparableBSSRDF Public Methods⟩ ≡

693

```
SeparableBSSRDF(const SurfaceInteraction &po, Float eta,
                  const Material *material, TransportMode mode)
: BSSRDF(po, eta), ns(po.shading.n), ss(Normalize(po.shading.dpdu)),
  ts(Cross(ns, ss)), material(material), mode(mode) { }
```

⟨SeparableBSSRDF Private Data⟩ ≡

693

```
const Normal3f ns;
const Vector3f ss, ts;
const Material *material;
const TransportMode mode;
```

The simplified SeparableBSSRDF interface casts the BSSRDF into a separable form with three independent components (one spatial and two directional):

$$S(p_o, \omega_o, p_i, \omega_i) \approx (1 - F_r(\cos \theta_o)) S_p(p_o, p_i) S_\omega(\omega_i). \quad [11.6]$$

The Fresnel term at the beginning models the fraction of the light that is transmitted into direction ω_o after exiting the material. A second Fresnel term contained inside $S_\omega(\omega_i)$ accounts for the influence of the boundary on the directional distribution of light entering the object from direction ω_i . The profile term S_p is a spatial distribution characterizing how far light travels after entering the material.

For high-albedo media, the scattered radiance distribution is generally fairly isotropic and the Fresnel transmittance is the most important factor for defining the final directional distribution. However, directional variation can be meaningful for low-albedo media; in that case, this approximation is less accurate.

(SeparableBSSRDF Public Methods) +≡ 693

```

Spectrum S(const SurfaceInteraction &pi, const Vector3f &wi) {
    Float Ft = 1 - FrDielectric(Dot(po.wo, po.shading.n), 1, eta);
    return Ft * Sp(pi) * Sw(wi);
}

```

Given the separable expression in Equation (11.6), the integral for determining the outgoing illumination due to subsurface scattering (Section 15.5) simplifies to

$$\begin{aligned} L_o(p_o, \omega_o) &= \int_A \int_{\mathcal{H}^2(n)} S(p_o, \omega_o, p_i, \omega_i) L_i(p_i, \omega_i) |\cos \theta_i| d\omega_i dA(p_i) \\ &= (1 - F_r(\cos \theta_o)) \int_A S_p(p_o, p_i) \int_{\mathcal{H}^2(n)} S_\omega(\omega_i) L_i(p_i, \omega_i) |\cos \theta_i| d\omega_i dA(p_i). \end{aligned}$$

We define the directional term $S_\omega(\omega_i)$ as a scaled version of the Fresnel transmittance (Section 8.2):

$$S_\omega(\omega_i) = \frac{1 - F_r(\cos \theta_i)}{c \pi}. \quad [11.7]$$

The normalization factor c is chosen so that S_ω integrates to one over the cosine-weighted hemisphere:

$$\int_{\mathcal{H}^2} S_\omega(\omega) \cos \theta d\omega = 1.$$

In other words,

$$\begin{aligned} c &= \int_0^{2\pi} \int_0^{\frac{\pi}{2}} \frac{1 - F_r(\eta, \cos \theta)}{\pi} \sin \theta \cos \theta d\theta d\phi \\ &= 1 - 2 \int_0^{\frac{\pi}{2}} F_r(\eta, \cos \theta) \sin \theta \cos \theta d\theta. \end{aligned}$$

This integral is referred to as the *first moment* of the Fresnel reflectance function. Other moments involving higher powers of the cosine function also exist and frequently occur in subsurface scattering-related computations; the general definition of the i th Fresnel moment is

$$\bar{F}_{r,i}(\eta) = \int_0^{\frac{\pi}{2}} F_r(\eta, \cos \theta) \sin \theta \cos^i \theta d\theta \quad [11.8]$$

BSSRDF::eta [692](#)
 BSSRDF::po [692](#)
 Dot() [63](#)
 Float [1062](#)
 FrDielectric() [519](#)
 Interaction::wo [115](#)
 SeparableBSSRDF::Sp() [695](#)
 SeparableBSSRDF::Sw() [695](#)
 Spectrum [315](#)
 SurfaceInteraction [116](#)
 SurfaceInteraction::shading::n [118](#)
 Vector3f [60](#)

pbrt provides two functions `FresnelMoment1()` and `FresnelMoment2()` that evaluate the corresponding moments based on polynomial fits to these functions. (We won't include their implementations in the book here.) One subtlety is that these functions follow a convention that is slightly different from the above definition—they are actually called with the reciprocal of η . This is due to their main application in Section 15.5, where they account for the effect light that is reflected back into the material due to a reflection at the internal boundary with a relative index of refraction of $1/\eta$.

(BSSRDF Utility Declarations) \equiv

```
Float FresnelMoment1(Float invEta);
Float FresnelMoment2(Float invEta);
```

Using `FresnelMoment1()`, the definition of `SeparableBSSRDF::Sw()` based on Equation (11.7) can be easily implemented.

(SeparableBSSRDF Public Methods) \equiv

693

```
Spectrum Sw(const Vector3f &w) const {
    Float c = 1 - 2 * FresnelMoment1(1 / eta);
    return (1 - FrDielectric(CosTheta(w), 1, eta)) / (c * Pi);
}
```

Decoupling the spatial and directional arguments considerably reduces the dimension of S but does not address the fundamental difficulty with regard to supporting general Shape implementations. We introduce a second approximation, which assumes that the surface is not only locally planar but that it is the distance between the points rather than their actual locations that affects the value of the BSSRDF. This reduces S_p to a function S_r that only involves distance of the two points p_o and p_i :

$$S_p(p_o, p_i) \approx S_r(\|p_o - p_i\|). \quad [11.9]$$

As before, the actual implementation of the spatial term S_p doesn't take p_o as an argument, since it is already available in `BSSRDF::po`.

(SeparableBSSRDF Public Methods) \equiv

693

```
Spectrum Sp(const SurfaceInteraction &pi) const {
    return Sr(Distance(po.p, pi.p));
}
```

The `SeparableBSSRDF::Sr()` method remains virtual—it is overridden in subclasses that implement specific 1D subsurface scattering profiles. Note that the dependence on the distance introduces an implicit assumption that the scattering medium is relatively homogeneous and does not strongly vary as a function of position—any variation should be larger than the mean free path length.

(SeparableBSSRDF Interface) \equiv

693

```
virtual Spectrum Sr(Float d) const = 0;
```

BSSRDF models are usually models of the function `Sr()` derived through a careful analysis of the light transport within a homogeneous slab. This means models such as the `SeparableBSSRDF` will yield good approximations in the planar setting, but with increasing error as the underlying geometry deviates from this assumption.

`BSSRDF::eta` 692
`BSSRDF::po` 692
`CosTheta()` 510
`Distance()` 70
`Float` 1062
`FrDielectric()` 519
`FresnelMoment1()` 695
`Interaction::p` 115
`Pi` 1063
`SeparableBSSRDF` 693
`SeparableBSSRDF::Sr()` 695
`SeparableBSSRDF::Sw()` 695
`Shape` 123
`Spectrum` 315
`SurfaceInteraction` 116
`Vector3f` 60



Figure 11.15: Objects rendered using the `TabulatedBSSRDF` using a variety of measured BSSRDFs. From left to right: cola, apple, skin, and ketchup.

11.4.2 TABULATED BSSRDF

```
(BSSRDF Declarations) +≡
class TabulatedBSSRDF : public SeparableBSSRDF {
public:
    (TabulatedBSSRDF Public Methods 697)
private:
    (TabulatedBSSRDF Private Data 697)
};
```

The single current implementation of the `SeparableBSSRDF` interface in `pbrt` is the `TabulatedBSSRDF` class. It provides access to a tabulated BSSRDF representation that can handle a wide range of scattering profiles including measured real-world BSSRDFs. `TabulatedBSSRDF` uses the same type of adaptive spline-based interpolation method that was also used by the `FourierBSDF` reflectance model (Section 8.6); in this case, we are interpolating the distance-dependent scattering profile function S_r from Equation (11.9). The `FourierBSDF`'s second stage of capturing directional variation using Fourier series is not needed. Figure 11.15 shows spheres rendered using the `TabulatedBSSRDF`.

It is important to note that the radial profile S_r is only a 1D function when all BSSRDF material properties are fixed. More generally, it depends on four additional parameters: the index of refraction η , the scattering anisotropy g , the albedo ρ , and the extinction coefficient σ_t , leading to a complete function signature $S_r(\eta, g, \rho, \sigma_t, r)$ that is unfortunately too high-dimensional for discretization. We must thus either remove or fix some of the parameters.

`FourierBSDF` 555
`SeparableBSSRDF` 693

Consider that the only parameter with physical units (apart from r) is σ_t . This parameter quantifies the rate of scattering or absorption interactions per unit distance. The effect of σ_t is simple: it only controls the spatial scale of the BSSRDF profile. To reduce the dimension of the necessary tables, we thus fix $\sigma_t = 1$ and tabulate a *unitless* version of the BSSRDF profile.

When a lookup for a given extinction coefficient σ_t and radius r occurs at run time we find the corresponding unitless *optical radius* $r_{\text{optical}} = \sigma_t r$ and evaluate the lower-dimensional tabulation as follows:

$$S_r(\eta, g, \rho, \sigma_t, r) = \sigma_t^2 S_r(\eta, g, \rho, 1, r_{\text{optical}}) \quad (11.10)$$

Since S_r is a 2D density function in polar coordinates (r, ϕ) , a corresponding scale factor of σ_t^2 is needed to account for this change of variables (see also Section 13.5.2).

We will also fix the index of refraction η and scattering anisotropy parameter g —in practice, this means that these parameters cannot be textured over an object that has a material that uses a `TabulatedBSSRDF`. These simplifications leave us with a fairly manageable 2D function that is only discretized over albedos ρ and optical radii r .

The `TabulatedBSSRDF` constructor takes all parameters of the `BSSRDF` constructor in addition to spectrally varying absorption and scattering coefficients σ_a and σ_s . It precomputes the extinction coefficient $\sigma_t = \sigma_a + \sigma_s$ and albedo $\rho = \sigma_s / \sigma_t$ for the current surface location, avoiding issues with a division by zero when there is no extinction for a given spectral channel.

(TabulatedBSSRDF Public Methods) ≡ 696

```
TabulatedBSSRDF(const SurfaceInteraction &po,
                  const Material *material, TransportMode mode, Float eta,
                  const Spectrum &sigma_a, const Spectrum &sigma_s,
                  const BSSRDFTable &table)
: SeparableBSSRDF(po, eta, material, mode), table(table) {
    sigma_t = sigma_a + sigma_s;
    for (int c = 0; c < Spectrum::nSamples; ++c)
        rho[c] = sigma_t[c] != 0 ? (sigma_s[c] / sigma_t[c]) : 0;
}
```

`BSSRDFTable` 697

`CoefficientSpectrum::nSamples` 318

`Float` 1062

`Material` 577

`SeparableBSSRDF::`
`SeparableBSSRDF()` 693

`Spectrum` 315

`SurfaceInteraction` 116

`TabulatedBSSRDF` 696

`TabulatedBSSRDF::rho` 697

`TabulatedBSSRDF::sigma_t` 697

`TabulatedBSSRDF::table` 697

`TransportMode` 960

(TabulatedBSSRDF Private Data) ≡ 696

```
const BSSRDFTable &table;
Spectrum sigma_t, rho;
```

Detailed information about the scattering profile S_r is supplied via the `table` parameter, which is an instance of the `BSSRDFTable` data structure:

(BSSRDF Declarations) +≡

```
struct BSSRDFTable {
    (BSSRDFTable Public Data 698)
    (BSSRDFTable Public Methods 698)
};
```

Instances of `BSSRDFTable` record samples of the function S_r taken at a set of single scattering albedos $(\rho_1, \rho_2, \dots, \rho_n)$ and radii (r_1, r_2, \dots, r_m) . The spacing between radius

and albedo samples will generally be non-uniform in order to more accurately represent the underlying function. Section 15.5.8 will show how to initialize the `BSSRDFTable` for a specific BSSRDF model.

We omit the constructor implementation, which takes the desired resolution and allocates memory for the representation.

(BSSRDFTable Public Methods) \equiv 697
`BSSRDFTable(int nRhoSamples, int nRadiusSamples);`

The sample locations and counts are exposed as public member variables.

(BSSRDFTable Public Data) \equiv 697
`const int nRhoSamples, nRadiusSamples;`
`std::unique_ptr<Float[]> rhoSamples, radiusSamples;`

A sample value is stored in the `profile` member variable for each of the $m \times n$ pairs (ρ_i, r_j) .

(BSSRDFTable Public Data) $+ \equiv$ 697
`std::unique_ptr<Float[]> profile;`

Note that the `TabulatedBSSRDF::rho` member variable gives the reduction in energy after a single scattering event; this is different from the material's overall albedo, which takes all orders of scattering into account. To stress the difference, we will refer to these different types of albedos as the *single scattering albedo* ρ and the *effective albedo* ρ_{eff} .

We define the effective albedo as the following integral of the profile S_r in polar coordinates.

$$\rho_{\text{eff}} = \int_0^{2\pi} \int_0^{\infty} r S_r(r) dr d\phi = 2\pi \int_0^{\infty} r S_r(r) dr. \quad [11.11]$$

The value of ρ_{eff} will frequently be accessed both by the profile sampling code and by the `KdSubsurfaceMaterial`. We introduce an array `rhoEff` of length `BSSRDFTable::nRhoSamples`, which maps every albedo sample to its corresponding effective albedo.

(BSSRDFTable Public Data) $+ \equiv$ 697
`std::unique_ptr<Float[]> rhoEff;`

Computation of ρ_{eff} will be discussed in Section 15.5. For now, we only note that it is a nonlinear and strictly monotonically increasing function of the single scattering albedo ρ .

Given radius value r and single scattering albedo, the function `Sr()` implements a spline-interpolated lookup into the tabulated profile. The albedo parameter is taken to be the `TabulatedBSSRDF::rho` value. There is thus an implicit assumption that the albedo does not vary within the support of the BSSRDF profile centered around p_0 .

Since the albedo is of type `Spectrum`, the function performs a separate profile lookup for each spectral channel and returns a `Spectrum` as a result. The return value must be clamped in case the interpolation produces slightly negative values.

`BSSRDFTable` [697](#)
`BSSRDFTable::nRhoSamples` [698](#)
`Float` [1062](#)
`KdSubsurfaceMaterial` [701](#)
`TabulatedBSSRDF::rho` [697](#)

```

⟨BSSRDF Method Definitions⟩ ≡
    Spectrum TabulatedBSSRDF::Sr(Float r) const {
        Spectrum Sr(0.f);
        for (int ch = 0; ch < Spectrum::nSamples; ++ch) {
            ⟨Convert r into unitless optical radius  $r_{\text{optical}}$  699⟩
            ⟨Compute spline weights to interpolate BSSRDF on channel ch 699⟩
            ⟨Set BSSRDF value Sr[ch] using tensor spline interpolation 699⟩
        }
        ⟨Transform BSSRDF value into world space units 700⟩
        return Sr.Clamp();
    }

```

The first line in the loop applies the scaling identity from Equation (11.10) to obtain an optical radius for the current channel ch .

```

⟨Convert r into unitless optical radius  $r_{\text{optical}}$ ⟩ ≡
    Float rOptical = r * sigma_t[ch];

```

699, 914

Given the adjusted radius rOptical and the albedo `TabulatedBSSRDF::rho[ch]` at location `BSSRDF::po`, we next call `CatmullRomWeights()` to obtain offsets and cubic spline weights to interpolate the profile values. This step is identical to the `FourierBSDF` interpolation in Section 8.6.

```

⟨Compute spline weights to interpolate BSSRDF on channel ch⟩ ≡
    int rhoOffset, radiusOffset;
    Float rhoWeights[4], radiusWeights[4];
    if (!CatmullRomWeights(table.nRhosamples, table.rhosamples.get(),
                           rho[ch], &rhoOffset, rhoWeights) ||
        !CatmullRomWeights(table.nRadiusSamples, table.radiusSamples.get(),
                           rOptical, &radiusOffset, radiusWeights))
        continue;

```

699

We can now sum over the product of the spline weights and the profile values.

```

⟨Set BSSRDF value Sr[ch] using tensor spline interpolation⟩ ≡
    Float sr = 0;
    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 4; ++j) {
            Float weight = rhoWeights[i] * radiusWeights[j];
            if (weight != 0)
                sr += weight * table.EvalProfile(rhoOffset + i,
                                                radiusOffset + j);
        }
    }
    ⟨Cancel marginal PDF factor from tabulated BSSRDF profile 700⟩
    Sr[ch] = sr;

```

699

A convenience method in `BSSRDFTable` helps with finding profile values.

```
(BSSRDFTable Public Methods) +≡ 697
    inline Float EvalProfile(int rhoIndex, int radiusIndex) const {
        return profile[rhoIndex * nRadiusSamples + radiusIndex];
    }
```

It's necessary to cancel a multiplicative factor of $2\pi r_{\text{optical}}$ that came from the entries of `BSSRDFTable::profile` related to Equation (11.11). This factor is present in the tabularized values to facilitate importance sampling (more about this in Section 15.4). Since it is not part of the definition of the BSSRDF, the term must be removed here.

```
(Cancel marginal PDF factor from tabulated BSSRDF profile) ≡ 699, 914
    if (rOptical != 0)
        sr /= 2 * Pi * rOptical;
```

Finally, we apply the change of variables factor from Equation (11.10) to convert the interpolated unitless BSSRDF value in `Sr` into world space units.

```
(Transform BSSRDF value into world space units) ≡ 699
    Sr *= sigma_t * sigma_t;
```

11.4.3 SUBSURFACE SCATTERING MATERIALS

There are two Materials for translucent objects: `SubsurfaceMaterial`, which is defined in `materials/subsurface.h` and `materials/subsurface.cpp`, and `KdSubsurfaceMaterial`, defined in `materials/kdsubsurface.h` and `materials/kdsubsurface.cpp`. The only difference between these two materials is how the scattering properties of the medium are specified.

```
(SubsurfaceMaterial Declarations) ≡
class SubsurfaceMaterial : public Material {
public:
    (SubsurfaceMaterial Public Methods)
private:
    (SubsurfaceMaterial Private Data 701)
};
```

`SubsurfaceMaterial` stores textures that allow the scattering properties to vary as a function of the position on the surface. Note that this isn't the same as scattering properties that vary as a function of 3D inside the scattering medium, but it can give a reasonable approximation to heterogeneous media in some cases. (Note, however, that if used with spatially varying textures, this feature destroys reciprocity of the BSSRDF, since these textures are evaluated at just one of the two scattering points, and so interchanging them will generally result in different values from the texture.⁵)

`BSSRDFTable` 697
`BSSRDFTable::nRadiusSamples` 698
`BSSRDFTable::profile` 698
`Float` 1062
`KdSubsurfaceMaterial` 701
`Material` 577
`Pi` 1063
`SubsurfaceMaterial` 700
`TabulatedBSSRDF::sigma_t` 697

⁵ A reciprocal version of this scheme could be obtained by averaging the albedos at p_i and p_o , though this is incompatible with the sampling scheme that is used currently.

In addition to the volumetric scattering properties, a number of textures allow the user to specify coefficients for a BSDF that represents perfect or glossy specular reflection and transmission at the surface.

```
<SubsurfaceMaterial Private Data> ≡ 700
    const Float scale;
    std::shared_ptr<Texture<Spectrum>> Kr, Kt, sigma_a, sigma_s;
    std::shared_ptr<Texture<Float>> uRoughness, vRoughness;
    std::shared_ptr<Texture<Float>> bumpMap;
    const Float eta;
    const bool remapRoughness;
    BSSRDFTable table;
```

The `ComputeScatteringFunctions()` method uses the textures to compute the values of the scattering properties at the point. The absorption and scattering coefficients are then scaled by the `scale` member variable, which provides an easy way to change the units of the scattering properties. (Recall that they're expected to be specified in terms of inverse meters.) Finally, the method creates a `TabulatedBSSRDF` with these parameters.

```
<SubsurfaceMaterial Method Definitions> ≡
void SubsurfaceMaterial::ComputeScatteringFunctions(
    SurfaceInteraction *si, MemoryArena &arena, TransportMode mode,
    bool allowMultipleLobes) const {
    <Perform bump mapping with bumpMap, if present 579>
    <Initialize BSDF for SubsurfaceMaterial>
    Spectrum sig_a = scale * sigma_a->Evaluate(*si).Clamp();
    Spectrum sig_s = scale * sigma_s->Evaluate(*si).Clamp();
    si->bssrdf = ARENA_ALLOC(arena, TabulatedBSSRDF)(
        *si, this, mode, eta, sig_a, sig_s, table);
}
```

BSSRDFTable 697

BxDF 513
 Float 1062
 MemoryArena 1074
 Spectrum 315
 Spectrum::Clamp() 317
 SubsurfaceFromDiffuse() 938
 SubsurfaceMaterial::scale 701
 SubsurfaceMaterial::sigma_a 701
 SubsurfaceMaterial::sigma_s 701
 SurfaceInteraction 116
 TabulatedBSSRDF 696
 Texture 614
 Texture::Evaluate() 615
 TransportMode 960

The fragment `<Initialize BSDF for SubsurfaceMaterial>` won't be included here; it follows the now familiar approach of allocating appropriate BxDF components for the BSDF according to which textures have nonzero SPDs.

Directly setting the absorption and scattering coefficients to achieve a desired visual look is difficult. The parameters have a nonlinear and unintuitive effect on the result. The `KdSubsurfaceMaterial` allows the user to specify the subsurface scattering properties in terms of the diffuse reflectance of the surface and the mean free path $1/\sigma_t$. It then uses the `SubsurfaceFromDiffuse()` utility function, which will be defined in Section 15.5, to compute the corresponding intrinsic scattering properties.

Being able to specify translucent materials in this manner is particularly useful in that it makes it possible to use standard texture maps that might otherwise be used for diffuse reflection to define scattering properties (again with the caveat that varying properties on the surface don't properly correspond to varying properties in the medium).

We won't include the definition of `KdSubsurfaceMaterial` here since its implementation just evaluates Textures to compute the diffuse reflection and mean free path values

and calls `SubsurfaceFromDiffuse()` to compute the scattering properties needed by the BSSRDF.

Finally, `GetMediumScatteringProperties()` is a utility function that has a small library of measured scattering data for translucent materials; it returns the corresponding scattering properties if it has an entry for the given name. (For a list of the valid names, see the implementation in `core/media.cpp`.) The data provided by this function is from papers by Jensen et al. (2001b) and Narasimhan et al. (2006).

(Media Declarations) +≡

```
bool GetMediumScatteringProperties(const std::string &name,
    Spectrum *sigma_a, Spectrum *sigma_s);
```

FURTHER READING

The books written by van de Hulst (1980) and Preisendorfer (1965, 1976) are excellent introductions to volume light transport. The seminal book by Chandrasekhar (1960) is another excellent resource, although it is mathematically challenging. See also the “Further Reading” section of Chapter 15 for more references on this topic.

The Henyey–Greenstein phase function was originally described by Henyey and Greenstein (1941). Detailed discussion of scattering and phase functions, along with derivations of phase functions that describe scattering from independent spheres, cylinders, and other simple shapes, can be found in van de Hulst’s book (1981). Extensive discussion of the Mie and Rayleigh scattering models is also available there. Hansen and Travis’s survey article is also a good introduction to the variety of commonly used phase functions (Hansen and Travis 1974).

While the Henyey–Greenstein model often works well, there are many materials that it can’t represent accurately. Gkioulekas et al. (2013b) showed that sums of Henyey–Greenstein and von Mises–Fisher lobes are more accurate for many materials than Henyey–Greenstein alone and derived a 2D parameter space that allows for intuitive control of translucent appearance.

Just as procedural modeling of textures is an effective technique for shading surfaces, procedural modeling of volume densities can be used to describe realistic-looking volumetric objects like clouds and smoke. Perlin and Hoffert (1989) described early work in this area, and the book by Ebert et al. (2003) has a number of sections devoted to this topic, including further references. More recently, accurate physical simulation of the dynamics of smoke and fire has led to extremely realistic volume data sets, including the ones used in this chapter; see, for example, Fedkiw, Stam, and Jensen (2001). See the book by Wrenninge (2012) for further information about modeling participating media, with particular focus on techniques used in modern feature film production.

In this chapter, we have ignored all issues related to sampling and antialiasing of volume density functions that are represented by samples in a 3D grid, although these issues should be considered, especially in the case of a volume that occupies just a few pixels on the screen. Furthermore, we have used a simple triangle filter to reconstruct densities at intermediate positions, which is suboptimal for the same reasons as the triangle filter

BSSRDF 692

Spectrum 315

`SubsurfaceFromDiffuse()` 938

is not a high-quality image reconstruction filter. Marschner and Lobb (1994) presented the theory and practice of sampling and reconstruction for 3D data sets, applying ideas similar to those in Chapter 7. See also the paper by Theußl, Hauser, and Gröller (2000) for a comparison of a variety of windowing functions for volume reconstruction with the sinc function and a discussion of how to derive optimal parameters for volume reconstruction filter functions.

Acquiring volumetric scattering properties of real-world objects is particularly difficult, requiring solving the inverse problem of determining the values that lead to the measured result. See Jensen et al. (2001b), Goesele et al. (2004), Narasimhan et al. (2006), and Peers et al. (2006) for recent work on acquiring scattering properties for subsurface scattering. More recently, Gkioulekas et al. (2013a) produced accurate measurements of a variety of media. Hawkins et al. (2005) have developed techniques to measure properties of media like smoke, acquiring measurements in real time. Another interesting approach to this problem was introduced by Frisvad et al. (2007), who developed methods to compute these properties from a lower-level characterization of the scattering properties of the medium.

Acquiring the volumetric density variation of participating media is also challenging. See work by Fuchs et al. (2007), Atcheson et al. (2008), and Gu et al. (2013a) for a variety of approaches to this problem, generally based on illuminating the medium in particular ways while photographing it from one or more viewpoints.

The medium representation used by `GridDensityMedium` doesn't adapt its spatial sampling rate as the amount of local detail in the underlying medium changes. Furthermore, its on-disk representation is a fairly inefficient string of floating-point values encoded as text. See Museth's VDB format (2013), or the Field3D system, which is described by Wrenninge (2015), for industrial-strength volume representation formats and libraries.

EXERCISES

- ② 11.1 Given a 1D volume density that is an arbitrary function of height $f(h)$, the optical distance between any two 3D points can be computed very efficiently if the integral $\int_0^{h'} f(h) dh$ is precomputed and stored in a table for a set of h' values (Perlin 1985b; Max 1986). Work through the mathematics to show the derivation for this approach, and implement it in `pbrt` by implementing a new `Medium` that takes an arbitrary function or a 1D table of density values. Compare the efficiency and accuracy of this approach to the default implementation of `Medium::Tr()`, which uses Monte Carlo integration.
- ② 11.2 The `GridDensityMedium` class uses a relatively large amount of memory for complex volume densities. Determine its memory requirements when used for the smoke images in this chapter, and modify its implementation to reduce memory use. One approach is to detect regions of space with constant (or relatively constant) density values using an octree data structure and to only refine the octree in regions where the densities are changing. Another possibility is to use less memory to record each density value, for example, by computing the minimum and maximum densities and then using 8 or 16 bits per density value

to interpolate between them. What sorts of errors appear when either of these approaches is pushed too far?

- ③ 11.3 Implement a new `Medium` that computes the scattering density at points in the medium procedurally—for example, by using procedural noise functions like those discussed in Section 10.6. You may find useful inspiration for procedural volume modeling primitives in Wrenninge’s book (2012).
- ③ 11.4 A shortcoming of a fully-procedural `Medium` like the one in Exercise 11.3 can be the inefficiency of evaluating the medium’s procedural functions repeatedly. Add a caching layer to a procedural medium that, for example, maintains a set of small regular voxel grids over regions of space. When a density lookup is performed, first check the cache to see if a value can be interpolated from one of the grids; otherwise update the cache to store the density function over a region of space that includes the lookup point. Study how many cache entries (and how much memory is consequently required) are needed for good performance. How do the cache size requirements change with volumetric path tracing that only accounts for direct lighting versus full global illumination? How do you explain this difference?

This page intentionally left blank



CHAPTER TWELVE

12 LIGHT SOURCES

In order for objects in a scene to be visible, there must be a source of illumination so that some light is reflected from them to the camera sensor. This chapter first describes different physical processes that lead to photon emission and then introduces the abstract Light class, which defines the interface used for light sources in pbrt. The implementations of a number of useful light sources follow. Because the implementations of different types of lights are all hidden behind a carefully designed interface, the light transport routines in Chapters 14, 15, and 16 can operate without knowing which particular types of lights are in the scene, similar to how acceleration structures can hold collections of different types of primitives without needing to know the details of their actual representations.

This chapter does not include implementations of all of the Light methods for all of the types of lights that are introduced. Many of the quantities related to complex light sources cannot be computed in closed form, and so Monte Carlo integration is needed. Therefore, the remainder of the Light methods are implemented in Section 14.2, after Monte Carlo methods has been introduced.

A wide variety of light source models are introduced in this chapter, although the variety is slightly limited by pbrt's physically based design. Many non-physical light source models have been developed for computer graphics, incorporating control over properties like the rate at which the light falls off with distance, which objects are illuminated by the light, which objects cast shadows from the light, and so on. These sorts of controls are incompatible with physically based light transport algorithms and thus can't be provided in the models here. As an example of the problems such lighting controls pose, consider a light that doesn't cast shadows: the total energy arriving at surfaces in the scene increases without bound as more surfaces are added. Consider a series of concentric shells of spheres around such a light; if occlusion is ignored, each added shell increases the total received energy. This directly violates the principle that the total energy arriving at surfaces illuminated by the light can't be greater than the total energy emitted by the light.

12.1 LIGHT EMISSION

All objects with temperature above absolute zero have moving atoms. In turn, as described by Maxwell's equations, the motion of atomic particles that hold electrical charges causes objects to emit electromagnetic radiation over a range of wavelengths. As we'll see shortly, most of the emission is at infrared frequencies for objects at room temperature; objects need to be much warmer to emit meaningful amounts of electromagnetic radiation at visible frequencies.

Many different types of light sources have been invented to convert energy into emitted electromagnetic radiation. Understanding some of the physical processes involved is helpful for accurately modeling light sources for rendering. A number are in wide use today:

- Incandescent (tungsten) lamps have a small tungsten filament. The flow of electricity through the filament heats it, which in turn causes it to emit electromagnetic radiation with a distribution of wavelengths that depends on the filament's temperature. A frosted glass enclosure is often present to absorb some of the wavelengths generated in order to achieve a desired SPD. With an incandescent light, much of the power in the SPD of the emitted electromagnetic radiation is in the infrared bands, which in turn means that much of the energy consumed by the light is turned into heat rather than light.
- Halogen lamps also have a tungsten filament, but the enclosure around them is filled with halogen gas. Over time, part of the filament in an incandescent light evaporates when it's heated; the halogen gas causes this evaporated tungsten to return to the filament, which lengthens the life of the light. Because it returns to the filament, the evaporated tungsten doesn't adhere to the bulb surface (as it does with regular incandescent bulbs), which also prevents the bulb from darkening.
- Gas-discharge lamps pass electrical current through hydrogen, neon, argon, or vaporized metal gas, which causes light to be emitted at specific wavelengths that depend on the particular atom in the gas. (Atoms that emit relatively little of their electromagnetic radiation in the not-useful infrared frequencies are selected for the gas.) Because a broader spectrum of wavelengths are generally more visually desirable than the chosen atoms generate directly, a fluorescent coating on the bulb's interior is often used to transform the emitted frequencies to a wider range. (The fluorescent coating also helps by converting ultraviolet wavelengths to visible wavelengths.)
- LED lights are based on electroluminescence: they use materials that emit photons due to electrical current passing through it.

For all of these sources, the underlying physical process is electrons colliding with atoms, which pushes their outer electrons to a higher energy level. When such an electron returns to a lower energy level, a photon is emitted. There are many other interesting processes that create light, including chemoluminescence (as seen in light sticks) and bioluminescence—a form of chemoluminescence seen in fireflies. Though interesting in their own right, we won't consider their mechanisms further here.

Luminous efficacy measures how effectively a light source converts power to visible illumination, accounting for the fact that for human observers, emission in non-visible

wavelengths is of little value. Interestingly enough, it is the ratio of a photometric quantity (the emitted luminous flux) to a radiometric quantity (either the total power it uses or the total power that it emits overall wavelengths, measured in flux):

$$\frac{\int \Phi_e(\lambda) V(\lambda) d\lambda}{\int \Phi_i(\lambda) d\lambda},$$

where $V(\lambda)$ is the spectral response curve introduced in Section 5.4.3.

Luminous efficacy has units of lumens per Watt. If Φ_i is the power consumed by the light source (rather than the emitted power), then luminous efficacy also incorporates a measure of how effectively the light source converts power to electromagnetic radiation. Luminous efficacy can also be defined as a ratio of luminous exitance (the photometric equivalent of radiant exitance) to irradiance at a point on the surface, or as the ratio of exitant luminance to radiance at a point on a surface in a particular direction.

A typical value of luminous efficacy for an incandescent tungsten lightbulb is around 15 lm/W. The highest value it can possibly have is 683, for a perfectly efficient light source that emits all of its light at $\lambda = 555$ nm, the peak of the $V(\lambda)$ function. (While such a light would have high efficacy, it wouldn't necessarily be a pleasant one as far as human observers are concerned.)

12.1.1 BLACKBODY EMITTERS

A *blackbody* is a perfect emitter: it converts power to electromagnetic radiation as efficiently as physically possible. While true blackbodies aren't physically realizable, some emitters exhibit near-blackbody behavior. Blackbodies also have a useful closed-form expression for their emission as a function of temperature and wavelength that is useful for modeling non-blackbody emitters.

Blackbodies are so-named because they absorb absolutely all incident power, reflecting none of it. Thus, an actual blackbody would appear perfectly black, no matter how much light was illuminating it. Intuitively, the reasons that perfect absorbers are also perfect emitters stem from the fact that absorption is the reverse operation of emission. Thus, if time was reversed, all of the perfectly absorbed power would be perfectly efficiently re-emitted.

Planck's law gives the radiance emitted by a blackbody as a function of wavelength λ and temperature T measured in Kelvins:

$$L_e(\lambda, T) = \frac{2hc^2}{\lambda^5 (e^{hc/\lambda k_b T} - 1)}, \quad [12.1]$$

where c is the speed of light in the medium (299, 792, 458 m/s in a vacuum), h is Planck's constant, $6.62606957 \times 10^{-34}$ J s, and k_b is the Boltzmann constant, $1.3806488 \times 10^{-23}$ J/K, where K is temperature in Kelvin. Blackbody emitters are perfectly diffuse; they emit radiance equally in all directions.

Figure 12.1 plots the emitted radiance distributions of a blackbody for a number of temperatures.

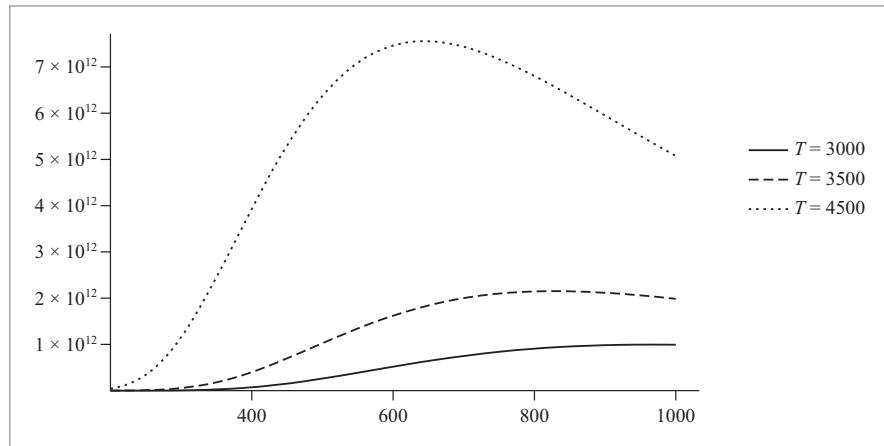


Figure 12.1: Plots of emitted radiance as a function of wavelength for blackbody emitters at a few temperatures, as given by Equation (12.1). Note that as temperature increases, more of the emitted light is in the visible frequencies (roughly 370 nm–730 nm) and that the spectral distribution shifts from reddish colors to bluish colors as the temperature increases. The total amount of emitted energy grows quickly as temperature increases, as described by the Stefan–Boltzmann law in Equation (12.2).

The `Blackbody()` function computes emitted radiance at the given temperature T in Kelvin for the n wavelengths in `lambda`.

(Spectrum Method Definitions) \equiv

```
void Blackbody(const Float *lambda, int n, Float T, Float *Le) {
    const Float c = 299792458;
    const Float h = 6.62606957e-34;
    const Float kb = 1.3806488e-23;
    for (int i = 0; i < n; ++i) {
        <Compute emitted radiance for blackbody at wavelength lambda[i] 710
    }
}
```

The `Blackbody()` function takes wavelengths in nm, but the constants for Equation (12.1) are in terms of meters. Therefore, we must first convert the wavelength to meters by scaling it by 10^{-9} .

(Compute emitted radiance for blackbody at wavelength lambda[i]) \equiv

710

```
Float l = lambda[i] * 1e-9;
Float lambda5 = (l * l) * (l * l) * l;
Le[i] = (2 * h * c * c) /
    (lambda5 * (std::exp((h * c) / (l * kb * T)) - 1));
```

The *Stefan–Boltzmann law* gives the radiant exitance (recall that this is the outgoing irradiance) at a point p for a blackbody emitter:

Float 1062

$$M(p) = \sigma T^4, \quad [12.2]$$

where σ is the Stefan–Boltzmann constant, $5.67032 \times 10^{-8} \text{ W m}^{-2} \text{ K}^{-4}$. Note that the total emission over all frequencies grows very rapidly—at the rate T^4 . Thus, doubling the temperature of a blackbody emitter increases the total energy emitted by a factor of 16.

Because the power emitted by a blackbody grows so quickly with temperature, it can also be useful to compute the normalized SPD for a blackbody where the maximum value of the SPD at any wavelength is 1. This is easily done with *Wien's displacement law*, which gives the wavelength where emission of a blackbody is maximum given its temperature:

$$\lambda_{\max} = \frac{b}{T}, \quad (12.3)$$

where b is Wien's displacement constant, $2.8977721 \times 10^{-3} \text{ m K}$.

(Spectrum Method Definitions) +≡

```
void BlackbodyNormalized(const Float *lambda, int n, Float T,
    Float *Le) {
    Blackbody(lambda, n, T, Le);
    (Normalize Le values based on maximum blackbody radiance 711)
}
```

Wien's displacement law gives the wavelength in meters where emitted radiance is at its maximum; we must convert this value to nm before calling `Blackbody()` to find the corresponding radiance value.

(Normalize Le values based on maximum blackbody radiance) ≡

```
Float lambdaMax = 2.8977721e-3 / T * 1e9;
Float maxL;
Blackbody(&lambdaMax, 1, T, &maxL);
for (int i = 0; i < n; ++i)
    Le[i] /= maxL;
```

711

The emission behavior of non-blackbodies is described by *Kirchoff's law*, which says that the emitted radiance distribution at any frequency is equal to the emission of a perfect blackbody at that frequency times the fraction of incident radiance at that frequency that is absorbed by the object. (This relationship follows from the object being assumed to be in thermal equilibrium.) The fraction of radiance absorbed is equal to 1 minus the amount reflected, and so the emitted radiance is

$$L'_e(T, \omega, \lambda) = L_e(T, \lambda)(1 - \rho_{hd}(\omega)), \quad (12.4)$$

where $L_e(T, \lambda)$ is the emitted radiance given by Planck's law, Equation (12.1), and $\rho_{hd}(\omega)$ is the hemispherical-directional reflectance from Equation (8.1).

The blackbody emission distribution provides as useful metric for describing the emission characteristics of non-blackbody emitters through the notion of *color temperature*. If the shape of the SPD of an emitter is similar to the blackbody distribution at some temperature, then we can say that the emitter has the corresponding color temperature. One approach to find color temperature is to take the wavelength where the light's emission is highest and find the corresponding temperature using Equation (12.3).

`Blackbody()` 710
`Float` 1062

Incandescent tungsten lamps are generally around 2700 K color temperature, and tungsten halogen lamps are around 3000 K. Fluorescent lights may range all the way from

2700 K to 6500 K. Generally speaking, color temperatures over 5000 K are described as “cool,” while 2700–3000 K is described as “warm.”

12.1.2 STANDARD ILLUMINANTS

Another useful way of categorizing light emission distributions are a number of “standard illuminants” that have been defined by Commission Internationale de l’Éclairage (CIE), which also specified the XYZ matching curves that we saw in Section 5.2.1.

The Standard Illuminant A was introduced in 1931 and was intended to represent average incandescent light. It corresponds to a blackbody radiator of about 2856 K. (It was originally defined as a blackbody at 2850 K, but the precision of the constants used in Planck’s law subsequently improved. Therefore, the specification was updated to be in terms of the 1931 constants, so that the illuminant was unchanged.) Figure 12.2 shows a plot of the SPD of the A illuminant.

(The B and C illuminants were intended to model daylight at two times of day and were generated with an A illuminant in combination with specific filters. They are no longer used. The E illuminant is defined as a constant-valued SPD and is used only for comparisons to other illuminants.)

The D illuminant describes various phases of daylight. It was defined based on characteristic vector analysis of a variety of daylight SPDs, which made it possible to express daylight in terms of a linear combination of three terms (one fixed and two weighted), with one weight essentially corresponding to yellow-blue color change due to cloudiness and the other corresponding to pink-green due to water in the atmosphere (from haze, etc.). D65 is roughly 6504 K color temperature (not 6500 K—again due to changes in the values used for the constants in Planck’s law) and is intended to correspond to mid-day

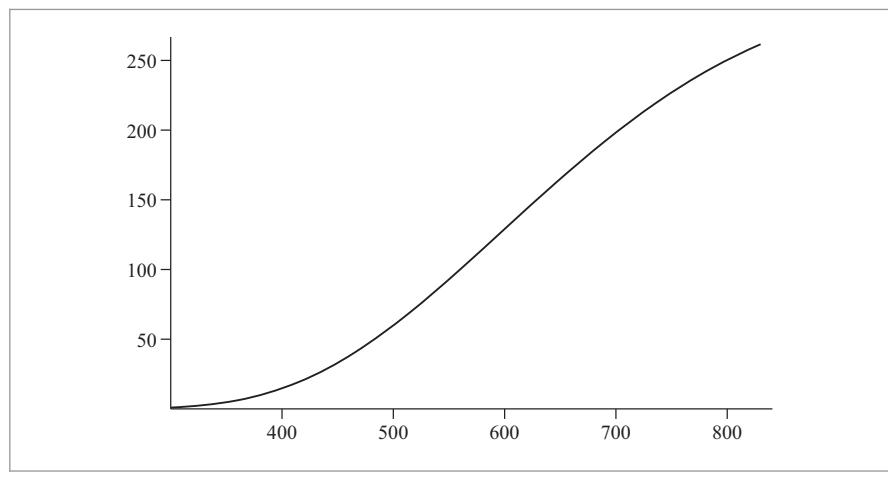


Figure 12.2: Plot of the CIE Standard Illuminant A’s SPD as a Function of Wavelength in nm.
This illuminant represents incandescent illumination and is close to a blackbody at 2856 K.

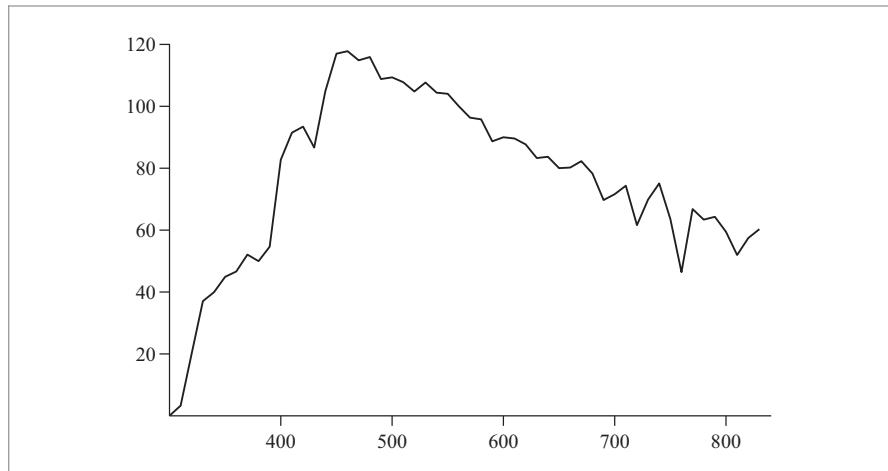


Figure 12.3: Plot of the CIE Standard Illuminant D's SPD as a Function of Wavelength in nm.
This illuminant represents noontime daylight in at European latitudes.

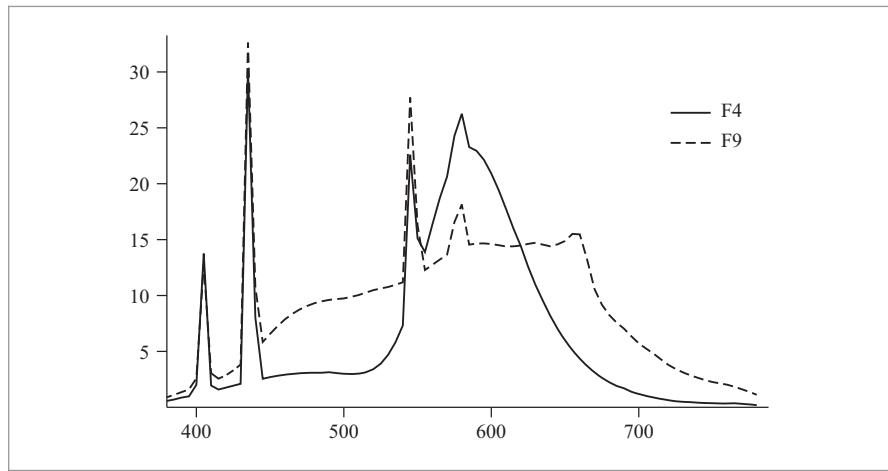


Figure 12.4: Plots of the F4 and F9 Standard Illuminants as a Function of Wavelength in nm.
These represent two fluorescent lights. Note that the SPDs are quite different. Spikes in the two distributions correspond to the wavelengths directly emitted by atoms in the gas, while the other wavelengths are generated by the bulb's fluorescent coating. The F9 illuminant is a "broadband" emitter that uses multiple phosphors to achieve a more uniform spectral distribution.

sunlight in Europe. (See Figure 12.3.) The CIE recommends that this illuminant be used for daylight unless there's a specific reason not to.

Finally, the F series of illuminants describes fluorescents; it is based on measurements of a number of actual fluorescent lights. Figure 12.4 shows the SPDs of two of them.

The files named `cie.stdillum.*` in the `scenes/spds` directory in the `pbrt` distribution have the SPDs of the standard illuminants, measured at 5-nm increments from 300 nm to 830 nm.

12.2 LIGHT INTERFACE

The core lighting routines and interfaces are in `core/light.h` and `core/light.cpp`. Implementations of particular lights are in individual source files in the `lights/` directory.

```
(Light Declarations) ≡
class Light {
public:
    (Light Interface 715)
    (Light Public Data 715)
protected:
    (Light Protected Data 715)
};
```

All lights share four common parameters:

1. The `flags` parameter indicates the fundamental light source type—for instance, whether or not the light is described by a delta distribution. (Examples of such lights include point lights, which emit illumination from a single point, and directional lights, where all light arrives from the same direction.) The Monte Carlo algorithms that sample illumination from light sources need to be aware of which lights are described by delta distributions, since this affects some of their computations.
2. A transformation that defines the light’s coordinate system with respect to world space. As with shapes, it’s often handy to be able to implement a light assuming a particular coordinate system (e.g., that a spotlight is always located at the origin of its light space, shining down the $+z$ axis). The light-to-world transformation makes it possible to place such lights at arbitrary positions and orientations in the scene.
3. A `MediumInterface` that describes the participating medium on the inside and the outside of the light source. For lights that don’t have “inside” and “outside” per se (e.g., a point light), the same `Medium` is on both sides. (A value of `nullptr` for both `Medium` pointers represents a vacuum.)
4. The `nSamples` parameter is used for area light sources where it may be desirable to trace multiple shadow rays to the light to compute soft shadows; it allows the user to have finer-grained control of the number of samples taken on a per-light basis. The default number of light source samples taken is 1; thus, only the light implementations for which taking multiple samples is sensible need to pass an explicit value to the `Light` constructor. Not all `Integrators` pay attention to this value.

`Integrator` 25

`Medium` 684

`MediumInterface` 684

The only other job for the constructor is to warn if the light-to-world transformation has a scale factor; many of the Light methods will return incorrect results in this case.¹

(Light Interface) ≡

```
Light(int flags, const Transform &LightToWorld,
      const MediumInterface &mediumInterface, int nSamples = 1)
: flags(flags), nSamples(std::max(1, nSamples)),
  mediumInterface(mediumInterface), LightToWorld(LightToWorld),
  WorldToLight(Inverse(LightToWorld)) {
  (Warn if light has transformation with non-uniform scale)
}
```

714

The flags, nSamples, and mediumInterface member variables are widely used outside of Light implementations so that it's worth making them available as public members.

(Light Public Data) ≡

```
const int flags;
const int nSamples;
const MediumInterface mediumInterface;
```

714

The LightFlags enumeration represents flags for the flags mask field characterizing various kinds of light sources; we'll see examples of all of these in the remainder of the chapter.

(LightFlags Declarations) ≡

```
enum class LightFlags : int {
    DeltaPosition = 1, DeltaDirection = 2, Area = 4, Infinite = 8
};
```

(LightFlags Declarations) +≡

```
inline bool IsDeltaLight(int flags) {
    return flags & (int)LightFlags::DeltaPosition ||
           flags & (int)LightFlags::DeltaDirection;
}
```

Interaction 115
Inverse() 1081
Light 714
Light::flags 715
Light::LightToWorld 715
Light::mediumInterface 715
Light::nSamples 715
Light::WorldToLight 715
LightFlags::DeltaDirection 715
LightFlags::DeltaPosition 715
MediumInterface 684
Transform 83

Although storing both the light-to-world and the world-to-light transformations is redundant, having both available simplifies code elsewhere by eliminating the need for calls for Inverse().

(Light Protected Data) ≡

```
const Transform LightToWorld, WorldToLight;
```

714

A key method that lights must implement is Sample_Li(). The caller passes an Interaction that provides the world space position of a reference point in the scene and a time associated with it, and the light returns the radiance arriving at that point at that time due to that light, assuming there are no occluding objects between them

¹ For example, the surface area reported by area lights is computed from the untransformed geometry, so a scale factor in the transformation means that the reported area and the actual area of the light in the scene would be inconsistent.

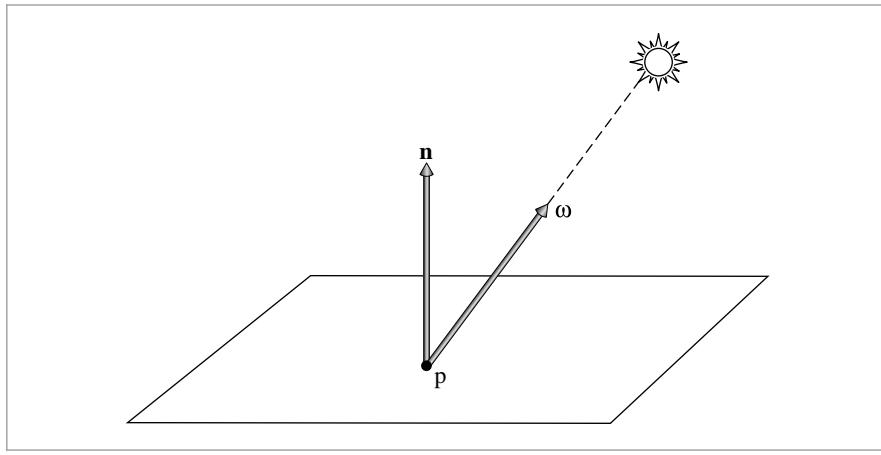


Figure 12.5: The `Light::Sample_Li()` method returns incident radiance from the light at a point p and also returns the direction vector ω_i that gives the direction from which radiance is arriving.

(Figure 12.5). Light implementations in pbrt do not currently support being animated—the lights themselves are at fixed positions in the scene. (Addressing this limitation is left as an exercise.) However, the time value from the `Interaction` is needed to set the time parameter in the traced visibility ray so that light visibility in the presence of moving objects is resolved correctly.

(Light Interface) +≡

714

```
virtual Spectrum Sample_Li(const Interaction &ref, const Point2f &u,
                           Vector3f *wi, Float *pdf, VisibilityTester *vis) const = 0;
```

The `Light` implementation is also responsible for both initializing the incident direction to the light source ω_i and initializing the `VisibilityTester` object, which holds information about the shadow ray that must be traced to verify that there are no occluding objects between the light and the reference point. The `VisibilityTester`, which will be described in Section 12.2.1, need not be initialized if the returned radiance value is black—for example, due to the reference point being outside of the cone of illumination of a spotlight. Visibility is irrelevant in this case.

For some types of lights, light may arrive at the reference point from many directions, not just from a single direction as with a point light source, for example. For these types of light sources, the `Sample_Li()` method samples a point on the light source's surface, so that Monte Carlo integration can be used to find the reflected light at the point due to illumination from the light. (The implementations of `Sample_Li()` for such lights will be introduced later, in Section 14.2.) The `Point2f u` parameter is used by these methods, and the `pdf` output parameter stores the probability density for the light sample that was taken. For all of the implementations in this chapter, the sample value is ignored and the `pdf` is set to 1. The `pdf` value's role in the context of Monte Carlo sampling is discussed in Section 14.2.

Float 1062
 Interaction 115
 Light 714
`Light::Sample_Li()` 716
 Point2f 68
 Spectrum 315
 Vector3f 60
 VisibilityTester 717

All lights must also be able to return their total emitted power; this quantity is useful for light transport algorithms that may want to devote additional computational resources to lights in the scene that make the largest contribution. Because a precise value for emitted power isn't needed elsewhere in the system, a number of the implementations of this method later in this chapter will compute approximations to this value rather than expending computational effort to find a precise value.

(Light Interface) +≡
 virtual Spectrum Power() const = 0;

714

Finally, Light interface includes a method Preprocess() that is invoked prior to rendering. It includes the Scene as an argument so that the light source can determine characteristics of the scene before rendering starts. The default implementation is empty, but some implementations (e.g., DistantLight) use it to record a bound of the scene extent.

(Light Interface) +≡
 virtual void Preprocess(const Scene &scene) { }

714

12.2.1 VISIBILITY TESTING

The VisibilityTester is a *closure*—an object that encapsulates a small amount of data and some computation that is yet to be done. It allows lights to return a radiance value under the assumption that the reference point and the light source are mutually visible. The integrator can then decide if illumination from the incident direction is relevant before incurring the cost of tracing the shadow ray—for example, light incident on the back side of a surface that isn't translucent contributes nothing to reflection from the other side. If the actual amount of arriving illumination is in fact needed, a call to one of the visibility tester's methods causes the necessary shadow ray to be traced.

(Light Declarations) +≡
 class VisibilityTester {
 public:
(VisibilityTester Public Methods 717)
 private:
 Interaction p0, p1;
};

VisibilityTesters are created by providing two Interaction objects, one for each end point of the shadow ray to be traced. Because an Interaction is used here, no special cases are needed for computing visibility to reference points on surfaces versus reference points in participating media.

(VisibilityTester Public Methods) ≡
 VisibilityTester(const Interaction &p0, const Interaction &p1)
 : p0(p0), p1(p1) { }

717

DistantLight 731
 Interaction 115
 Scene 23
 Spectrum 315
 VisibilityTester 717
 VisibilityTester::p0 717
 VisibilityTester::p1 717

Some of the light transport routines find it useful to be able to retrieve the two end points from an initialized VisibilityTester.

(VisibilityTester Public Methods) +≡

```
const Interaction &P0() const { return p0; }
const Interaction &P1() const { return p1; }
```

717

There are two methods that determine the visibility between the two points. The first, `Unoccluded()`, traces a shadow ray between them and returns a Boolean result. Some ray tracers include a facility for casting colored shadows from partially transparent objects and would return a spectrum from a method like this, but `pbtr` does not include this facility, since this feature generally requires a nonphysical hack. Scenes where illumination passes through a transparent object should be rendered with an integrator that supports this kind of effect; any of the bidirectional integrators from Chapter 16 is a good choice.

(Light Method Definitions) ≡

```
bool VisibilityTester::Unoccluded(const Scene &scene) const {
    return !scene.IntersectP(p0.SpawnRayTo(p1));
}
```

Because it only returns a Boolean value, `Unoccluded()` also ignores the effects of any scattering medium that the ray passes through on the radiance that it carries. When Integrators need to account for that effect, they use the `VisibilityTester`'s `Tr()` method instead. `VisibilityTester::Tr()` computes the beam transmittance, Equation (11.1), the fraction of radiance transmitted along the segment between the two points. It accounts for both attenuation in participating media as well as any surfaces that block the ray completely.

(Light Method Definitions) +≡

```
Spectrum VisibilityTester::Tr(const Scene &scene,
                               Sampler &sampler) const {
    Ray ray(p0.SpawnRayTo(p1));
    Spectrum Tr(1.f);
    while (true) {
        SurfaceInteraction isect;
        bool hitSurface = scene.Intersect(ray, &isect);
        <Handle opaque surface along ray's path 718>
        <Update transmittance for current ray segment 719>
        <Generate next ray segment or return final transmittance 719>
    }
    return Tr;
}
```

Interaction 115
`Interaction::SpawnRayTo()` 232
`Primitive::GetMaterial()` 249
`Ray` 73
`Sampler` 421
`Scene` 23
`Scene::Intersect()` 24
`Scene::IntersectP()` 24
`Spectrum` 315
`SurfaceInteraction` 116
`SurfaceInteraction::primitive` 249
`VisibilityTester` 717
`VisibilityTester::p0` 717
`VisibilityTester::p1` 717

If an intersection is found along the ray segment and the hit surface is opaque, then the ray is blocked and the transmittance is zero. Our work here is done. (Recall from Section 11.3 that surfaces with a `nullptr` material pointer should be ignored in ray visibility tests, as those surfaces are only used to bound the extent of participating media.)

<Handle opaque surface along ray's path> ≡

718

```
if (hitSurface && isect.primitive->GetMaterial() != nullptr)
    return Spectrum(0.0f);
```

Otherwise, the `Tr()` method accumulates the ray's transmittance, either to the surface intersection point or to the endpoint `p1`. (If there was an intersection with a non-opaque surface, the `Ray::tMax` value has been updated accordingly; otherwise it corresponds to `p1`.) In either case, `Medium::Tr()` computes the beam transmittance up to `Ray::tMax`, using the multiplicative property of beam transmittance from Equation (11.2).

```
{Update transmittance for current ray segment} ≡
if (ray.medium)
    Tr *= ray.medium->Tr(ray, sampler);
```

718

If no intersection was found, the ray made it to `p1` and we've accumulated the full transmittance. Otherwise, the ray intersected an invisible surface and the loop runs again, tracing a ray from that intersection point onward toward `p1`.

```
{Generate next ray segment or return final transmittance} ≡
if (!hitSurface)
    break;
ray = isect.SpawnRayTo(p1);
```

718

12.3 POINT LIGHTS

A number of interesting lights can be described in terms of emission from a single point in space with some possibly angularly varying distribution of outgoing light. This section describes the implementation of a number of them, starting with `PointLight`, which represents an isotropic point light source that emits the same amount of light in all directions. It is defined in `lights/point.h` and `lights/point.cpp`. Figure 12.6 shows a scene rendered with a point light source. Building on this base, a number of more complex lights based on point sources will be introduced, including spotlights and a light that projects an image into the scene.

```
{PointLight Declarations} ≡
class PointLight : public Light {
public:
    {PointLight Public Methods 720}
private:
    {PointLight Private Data 720}
};
```

`Interaction::SpawnRayTo()`
232
`Light` 714
`Light::flags` 715
`LightFlags::DeltaPosition`
715
`Medium::Tr()` 684
`PointLight` 719
`Ray::medium` 74
`Ray::tMax` 73
`VisibilityTester::p1` 717

`PointLights` are positioned at the origin in light space. To place them elsewhere, the light-to-world transformation should be set appropriately. Using this transformation, the world space position of the light is precomputed and cached in the constructor by transforming $(0, 0, 0)$ from light space to world space.

The constructor also stores the intensity for the light source, which is the amount of power per unit solid angle. Because the light source is isotropic, this is a constant. Finally, since point lights represent singularities that only emit light from a single position, the `Light::flags` field is initialized to `LightFlags::DeltaPosition`.



Figure 12.6: Scene Rendered with a Point Light Source. Notice the hard shadow boundaries from this type of light.

(PointLight Public Methods) ≡

719

```
PointLight(const Transform &LightToWorld,
           const MediumInterface &mediumInterface, const Spectrum &I)
: Light((int)LightFlags::DeltaPosition, LightToWorld,
       mediumInterface),
pLight(LightToWorld(Point3f(0, 0, 0))), I(I) { }
```

(PointLight Private Data) ≡

719

```
const Point3f pLight;
const Spectrum I;
```

Strictly speaking, it is incorrect to describe the light arriving at a point due to a point light source using units of radiance. Radiant intensity is instead the proper unit for describing emission from a point light source, as explained in Section 5.4. In the light source interfaces here, however, we will abuse terminology and use `Sample_Li()` methods to report the illumination arriving at a point for all types of light sources, dividing radiant intensity by the squared distance to the point p to convert units. Section 14.2 revisits the details of this issue in its discussion of how delta distributions affect evaluation of the integral in the scattering equation. In the end, the correctness of the computation does not suffer from this fudge, and it makes the implementation of light transport algorithms more straightforward by not requiring them to use different interfaces for different types of lights.

Light 714

LightFlags::DeltaPosition
715

MediumInterface 684

Point3f 68

Pointlight 719

PointLight::I 720

PointLight::pLight 720

Spectrum 315

Transform 83

```
<PointLight Method Definitions> ≡
    Spectrum PointLight::Sample_Li(const Interaction &ref,
        const Point2f &u, Vector3f *wi, Float *pdf,
        VisibilityTester *vis) const {
        *wi = Normalize(pLight - ref.p);
        *pdf = 1.f;
        *vis = VisibilityTester(ref, Interaction(pLight, ref.time,
            mediumInterface));
        return I / DistanceSquared(pLight, ref.p);
    }
```

The total power emitted by the light source can be found by integrating the intensity over the entire sphere of directions:

$$\Phi = \int_{S^2} I \, d\omega = I \int_{S^2} d\omega = 4\pi I.$$

```
<PointLight Method Definitions> +≡
    Spectrum PointLight::Power() const {
        return 4 * Pi * I;
    }
```

12.3.1 SPOTLIGHTS

Spotlights are a handy variation on point lights; rather than shining illumination in all directions, they emit light in a cone of directions from their position. For simplicity, we will define the spotlight in the light coordinate system to always be at position (0, 0, 0) and pointing down the +z axis. To place or orient it elsewhere in the scene, the Light::WorldToLight transformation should be set accordingly. Figure 12.7 shows a rendering of the same scene as Figure 12.6, only illuminated with a spotlight instead of a point light. The SpotLight class is defined in `lights/spot.h` and `lights/spot.cpp`.

DistanceSquared() 70
 Float 1062
 Interaction 115
 Interaction::p 115
 Interaction::time 115
 Light 714
 Light::mediumInterface 715
 Light::WorldToLight 715
 Pi 1063
 Point2f 68
 PointLight 719
 PointLight::I 720
 PointLight::pLight 720
 Spectrum 315
 SpotLight 721
 Vector3f::Normalize() 66
 Vector3f 60
 VisibilityTester 717

```
<SpotLight Declarations> ≡
    class SpotLight : public Light {
    public:
        (SpotLight Public Methods)
    private:
        (SpotLight Private Data 723)
    };
```

Two angles are passed to the constructor to set the extent of the SpotLight's cone: the overall angular width of the cone and the angle at which falloff starts (Figure 12.8). The constructor precomputes and stores the cosines of these angles for use in the SpotLight's methods.



Figure 12.7: Scene Rendered with a Spotlight. The spotlight cone smoothly cuts off illumination past a user-specified angle from the light's central axis.

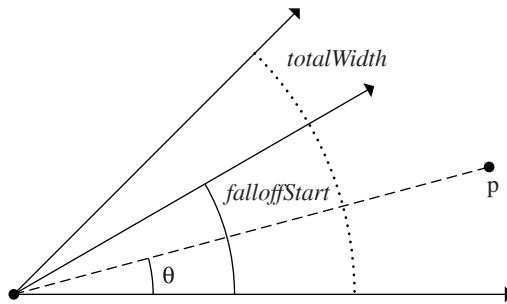


Figure 12.8: Spotlights are defined by two angles, *falloffStart* and *totalWidth*. Objects inside the inner cone of angles, up to *falloffStart*, are fully illuminated by the light. The directions between *falloffStart* and *totalWidth* are a transition zone that ramps down from full illumination to no illumination, such that points outside the *totalWidth* cone aren't illuminated at all. The cosine of the angle between the vector to a point p and the spotlight axis, θ , can easily be computed with a dot product.

```
<SpotLight Method Definitions> ≡
    SpotLight::SpotLight(const Transform &LightToWorld,
        const MediumInterface &mediumInterface, const Spectrum &I,
        Float totalWidth, Float falloffStart)
    : Light((int)LightFlags::DeltaPosition, LightToWorld,
        mediumInterface),
        pLight(LightToWorld(Point3f(0, 0, 0))), I(I),
        cosTotalWidth(std::cos(Radians(totalWidth))),
        cosFalloffStart(std::cos(Radians(falloffStart))) { }
```

<SpotLight Private Data> ≡

```
const Point3f pLight;
const Spectrum I;
const Float cosTotalWidth, cosFalloffStart;
```

721

The `SpotLight::Sample_Li()` method is almost identical to `PointLight::Sample_Li()`, except that it also calls the `Falloff()` method, which computes the distribution of light accounting for the spotlight cone. This computation is encapsulated in a separate method since other `SpotLight` methods will need to perform it as well.

```
DistanceSquared() 70
Float 1062
Interaction 115
Interaction::p 115
Interaction::time 115
Light 714
Light::LightToWorld 715
Light::mediumInterface 715
LightFlags::DeltaPosition 715
MediumInterface 684
Point2f 68
Point3f 68
PointLight::Sample_Li() 721
Radians() 1063
Spectrum 315
SpotLight 721
SpotLight::cosFalloffStart 723
SpotLight::cosTotalWidth 723
SpotLight::Falloff() 724
SpotLight::I 723
SpotLight::pLight 723
SpotLight::Sample_Li() 723
Transform 83
Vector3f::Normalize() 66
Vector3f 60
VisibilityTester 717
```

<SpotLight Method Definitions> +≡

```
Spectrum SpotLight::Sample_Li(const Interaction &ref,
    const Point2f &u, Vector3f *wi, Float *pdf,
    VisibilityTester *vis) const {
    *wi = Normalize(pLight - ref.p);
    *pdf = 1.f;
    *vis = VisibilityTester(ref, Interaction(pLight, ref.time,
        mediumInterface));
    return I * Falloff(-*wi) / DistanceSquared(pLight, ref.p);
}
```

To compute the spotlight's strength for a receiving point p , this first step is to compute the cosine of the angle between the vector from the spotlight origin to p and the vector along the center of the spotlight's cone. To compute the cosine of the offset angle to a point p , we have, as illustrated in Figure 12.8,

$$\begin{aligned}\cos \theta &= (\hat{p} \cdot (0, 0, 0)) \cdot (0, 0, 1) \\ &= p_z / \|p\|.\end{aligned}$$

This value is then compared to the cosines of the falloff and overall width angles to see where the point lies with respect to the spotlight cone. We can trivially determine that points with a cosine greater than the cosine of the falloff angle are inside the cone receiving full illumination, and points with cosine less than the width angle's cosine are completely outside the cone. (Note that the computation is slightly tricky since for $\theta \in [0, \pi]$, if $\theta > \theta'$, then $\cos \theta < \cos \theta'$.)

```
(SpotLight Method Definitions) +≡
    Float SpotLight::Falloff(const Vector3f &w) const {
        Vector3f wl = Normalize(WorldToLight(w));
        Float cosTheta = wl.z;
        if (cosTheta < cosTotalWidth)      return 0;
        if (cosTheta > cosFalloffStart)   return 1;
        {Compute falloff inside spotlight cone 724}
    }
```

For points inside the transition range from fully illuminated to outside of the cone, the intensity is scaled to smoothly fall off from full illumination to darkness:²

```
(Compute falloff inside spotlight cone) ≡ 724
    Float delta = (cosTheta - cosTotalWidth) /
                  (cosFalloffStart - cosTotalWidth);
    return (delta * delta) * (delta * delta);
```

The solid angle subtended by a cone with spread angle θ is $2\pi(1 - \cos \theta)$. Therefore, the integral over directions on the sphere that gives power from radiant intensity can be solved to compute the total power of a light that only emits illumination in a cone. For the spotlight, we can reasonably approximate the power of the light by computing the solid angle of directions that is covered by the cone with a spread angle cosine halfway between width and fall.

```
(SpotLight Method Definitions) +≡
    Spectrum SpotLight::Power() const {
        return I * 2 * Pi * (1 - .5f * (cosFalloffStart + cosTotalWidth));
    }
```

12.3.2 TEXTURE PROJECTION LIGHTS

Another useful light source acts like a slide projector; it takes an image map and projects its image out into the scene. The `ProjectionLight` class uses a projective transformation to project points in the scene onto the light's projection plane based on the field of view angle given to the constructor (Figure 12.9). Its implementation is in `lights/projection.h` and `lights/projection.cpp`. The use of this light in the lighting example scene is shown in Figure 12.10.

```
(ProjectionLight Declarations) ≡
    class ProjectionLight : public Light {
    public:
        (ProjectionLight Public Methods)
    private:
        (ProjectionLight Private Data 726)
    };
```

Float 1062
Light 714
Light::WorldToLight 715
Pi 1063
ProjectionLight 724
Spectrum 315
SpotLight 721
SpotLight::cosFalloffStart 723
SpotLight::cosTotalWidth 723
SpotLight::I 723
Vector3f::Normalize() 66
Vector3f 60

² Note the parentheses in the expression that takes `delta` to the fourth power: this allows the compiler to generate two multiply instructions rather than three instructions, as would be needed if the parentheses were removed. Consider the discussion of IEEE floating point in Section 3.9.1 to see why the parentheses are necessary for this.

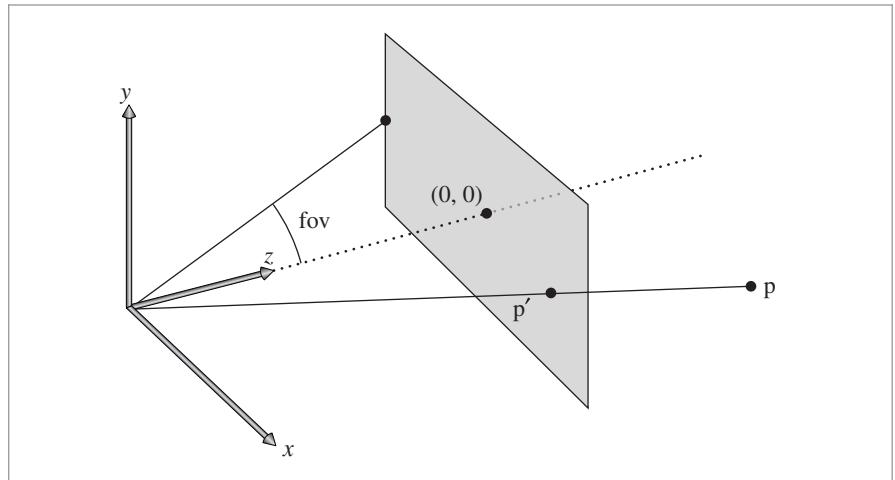


Figure 12.9: The Basic Setting for Projection Light Sources. A point p in the light's coordinate system is projected onto the plane of the image using the light's projection matrix.

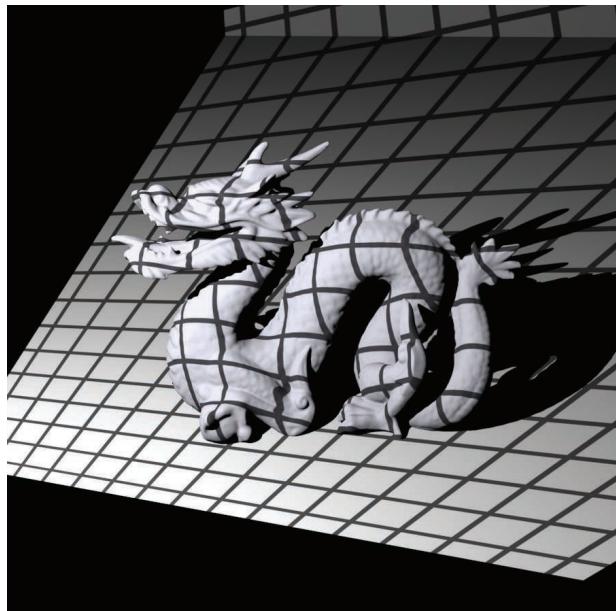


Figure 12.10: Scene Rendered with a Projection Light Using a Grid Texture Map. The projection light acts like a slide projector, projecting an image onto objects in the scene.

```
(ProjectionLight Method Definitions) ≡
ProjectionLight::ProjectionLight(const Transform &LightToWorld,
    const MediumInterface &mediumInterface, const Spectrum &I,
    const std::string &texname, Float fov)
: Light((int)LightFlags::DeltaPosition, LightToWorld,
    mediumInterface),
    pLight(LightToWorld(Point3f(0, 0, 0))), I(I) {
(Create ProjectionLight MIP map 726)
(Initialize ProjectionLight projection matrix 726)
(Compute cosine of cone surrounding projection directions 727)
}
```

This light could use a Texture to represent the light projection distribution so that procedural projection patterns could be used. However, having a precise representation of the projection function, as is available by using an image in a MIPMap, is useful for being able to sample the projection distribution using Monte Carlo techniques, so we will use that representation in the implementation here.

Note that the projected image is explicitly stored as an RGBSpectrum, even if full spectral rendering is being performed. Unless the image map being used has full spectral data, storing full SampledSpectrum values in this case only wastes memory; whether an RGB color is converted to a SampledSpectrum before the MIPMap is created or after the MIPMap returns a value from its Lookup() routine gives the same result in either case.

```
(Create ProjectionLight MIP map) ≡ 726
Point2i resolution;
std::unique_ptr<RGBSpectrum[]> texels = ReadImage(texname, &resolution);
if (texels)
    projectionMap.reset(new MIPMap<RGBSpectrum>(resolution,
        texels.get()));

(ProjectionLight Private Data) ≡ 724
std::unique_ptr<MIPMap<RGBSpectrum>> projectionMap;
const Point3f pLight;
const Spectrum I;

Similar to the PerspectiveCamera, the ProjectionLight constructor computes a projection matrix and the screen space extent of the projection.

(Initialize ProjectionLight projection matrix) ≡ 726
Float aspect = projectionMap ?
    (Float(resolution.x) / Float(resolution.y)) : 1;
if (aspect > 1)
    screenBounds = Bounds2f(Point2f(-aspect, -1), Point2f(aspect, 1));
else
    screenBounds = Bounds2f(Point2f(-1, -1/aspect), Point2f(1, 1/aspect));
near = 1e-3f;
far = 1e30f;
lightProjection = Perspective(fov, near, far);

Keywords
```

- Bounds2f 76
- Float 1062
- Light 714
- Light::LightToWorld 715
- LightFlags::DeltaPosition 715
- MediumInterface 684
- MIPMap 625
- Perspective() 365
- PerspectiveCamera 365
- Point2f 68
- Point2i 68
- Point3f 68
- ProjectionLight 724
- ProjectionLight::far 727
- ProjectionLight::I 726
- ProjectionLight::
 lightProjection 727
- ProjectionLight::near 727
- ProjectionLight::pLight 726
- ProjectionLight::
 projectionMap 726
- ProjectionLight::screenBounds 727
- ReadImage() 1067
- RGBSpectrum 332
- SampledSpectrum 319
- Spectrum 315
- Texture 614
- Transform 83

(ProjectionLight Private Data) +≡

```
    Transform lightProjection;
    Float near, far;
    Bounds2f screenBounds;
```

724

Finally, the constructor finds the cosine of the angle between the $+z$ axis and the vector to a corner of the screen window. This value is used elsewhere to define the minimal cone of directions that encompasses the set of directions in which light is projected. This cone is useful for algorithms like photon mapping that need to randomly sample rays leaving the light source (explained in Chapter 16). We won't derive this computation here; it is based on straightforward trigonometry.

(Compute cosine of cone surrounding projection directions) ≡

```
    Float opposite = std::tan(Radians(fov) / 2.f);
    Float tanDiag = opposite * std::sqrt(1 + 1 / (aspect * aspect));
    cosTotalWidth = std::cos(std::atan(tanDiag));
```

726

(ProjectionLight Private Data) +≡

```
    Float cosTotalWidth;
```

724

Similar to the spotlight's version, `ProjectionLight::Sample_Li()` calls a utility method, `ProjectionLight::Projection()`, to determine how much light is projected in the given direction. Therefore, we won't include the implementation of `Sample_Li()` here.

(ProjectionLight Method Definitions) +≡

```
Spectrum ProjectionLight::Projection(const Vector3f &w) const {
    Vector3f wl = WorldToLight(w);
    (Discard directions behind projection light 727)
    (Project point onto projection plane and compute light 728)
}
```

Because the projective transformation has the property that it projects points behind the center of projection to points in front of it, it is important to discard points with a negative z value. Therefore, the projection code immediately returns no illumination for projection points that are behind the near plane for the projection. If this check were not done, then it wouldn't be possible to know if a projected point was originally behind the light (and therefore not illuminated) or in front of it.

(Discard directions behind projection light) ≡

```
    if (wl.z < near) return 0;
```

727

After being projected to the projection plane, points with coordinate values outside the screen window are discarded. Points that pass this test are transformed to get (s, t) texture coordinates inside $[0, 1]^2$ for the lookup in the image map. Note it is explicitly specified that the `RGBSpectrum` value passed to the `Spectrum` constructor represents an illuminant's SPD and not that of a reflectance. (Recall from Section 5.2.2 that different matching functions are used for converting from RGB to SPDs for illuminants versus reflectances.)

(Project point onto projection plane and compute light) ≡ 727

```
Point3f p = lightProjection(Point3f(wl.x, wl.y, wl.z));
if (!Inside(Point2f(p.x, p.y), screenBounds)) return 0.f;
if (!projectionMap) return 1;
Point2f st = Point2f(screenBounds.Offset(Point2f(p.x, p.y)));
return Spectrum(projectionMap->Lookup(st), SpectrumType::Illuminant);
```

The total power of this light is approximated as a spotlight that subtends the same angle as the diagonal of the projected image, scaled by the average intensity in the image map. This approximation becomes increasingly inaccurate as the projected image's aspect ratio becomes less square, for example, and it doesn't account for the fact that texels toward the edges of the image map subtend a larger solid angle than texels in the middle when projected with a perspective projection. Nevertheless, it's a reasonable first-order approximation.

(ProjectionLight Method Definitions) +≡

```
Spectrum ProjectionLight::Power() const {
    return (projectionMap ?
        Spectrum(projectionMap->Lookup(Point2f(.5f, .5f), .5f),
                 SpectrumType::Illuminant) : Spectrum(1.f)) *
        I * 2 * Pi * (1.f - cosTotalWidth);
}
```

12.3.3 GONIOPHOTOMETRIC DIAGRAM LIGHTS

A *goniophotometric diagram* describes the angular distribution of luminance from a point light source; it is widely used in illumination engineering to characterize lights. Figure 12.11 shows an example of a goniophotometric diagram in two dimensions. In this section, we'll implement a light source that uses goniophotometric diagrams encoded in 2D image maps to describe the emission distribution of the light. The implementation is very similar to the point light sources defined previously in this section; it scales the intensity based on the outgoing direction according to the goniophotometric diagram's values. Figure 12.12 shows a few goniophotometric diagrams encoded as image maps, and Figure 12.13 shows a scene rendered with a light source that uses one of these images to modulate its directional distribution of illumination.

(GonioPhotometricLight Declarations) ≡

```
class GonioPhotometricLight : public Light {
public:
    (GonioPhotometricLight Public Methods 730)
private:
    (GonioPhotometricLight Private Data 730)
};
```

The GonioPhotometricLight constructor takes a base intensity and an image map that scales the intensity based on the angular distribution of light.

```
Bounds2::Inside() 79
GonioPhotometricLight 728
Light 714
MIPMap::Lookup() 635
Pi 1063
Point2f 68
Point3f 68
ProjectionLight::
    cosTotalWidth
    727
ProjectionLight::I 726
ProjectionLight::
    lightProjection
    727
ProjectionLight::
    projectionMap
    726
ProjectionLight::screenBounds
    727
Spectrum 315
SpectrumType::Illuminant 330
```

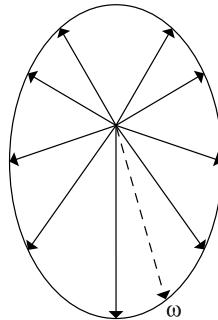


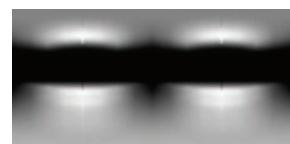
Figure 12.11: An Example of a Goniophotometric Diagram Specifying an Outgoing Light Distribution from a Point Light Source in 2D. The emitted intensity is defined in a fixed set of directions on the unit sphere, and the intensity for a given outgoing direction ω is found by interpolating the intensities of the adjacent samples.



(a)



(b)



(c)

Figure 12.12: Goniophotometric Diagrams for Real-World Light Sources, Encoded as Image Maps with a Parameterization Based on Spherical Coordinates. (a) A light that mostly illuminates in its up direction, with only a small amount of illumination in the down direction. (b) A light that mostly illuminates in the down direction. (c) A light that casts illumination both above and below.



Figure 12.13: Scene Rendered Using a Goniophotometric Diagram from Figure 12.12. Even though a point light source is the basis of this light, including the directional variation of a realistic light improves the visual realism of the rendered image.

```
(GonioPhotometricLight Public Methods) ≡ 728
GonioPhotometricLight(const Transform &LightToWorld,
    const MediumInterface &mediumInterface, const Spectrum &I,
    const std::string &texname)
: Light((int)LightFlags::DeltaPosition, LightToWorld,
    mediumInterface),
pLight(LightToWorld(Point3f(0, 0, 0))), I(I) {
{Create mipmap for GonioPhotometricLight 730}
}

(GonioPhotometricLight Private Data) ≡ 728
const Point3f pLight;
const Spectrum I;
std::unique_ptr<MIPMap<RGBSpectrum>> mipmap;

Like ProjectionLight, GonioPhotometricLight constructs a MIPMap of the distribution's
image map, also always using RGBSpectrum values.

(Create mipmap for GonioPhotometricLight) ≡ 730
Point2i resolution;
std::unique_ptr<RGBSpectrum[]> texels = ReadImage(texname, &resolution);
if (texels)
    mipmap.reset(new MIPMap<RGBSpectrum>(resolution, texels.get()));

The GonioPhotometricLight::Sample_Li() method is not shown here. It is essentially
identical to the SpotLight::Sample_Li() and ProjectionLight::Sample_Li() methods
that use a helper function to scale the amount of radiance. It assumes that the scale
texture is encoded using spherical coordinates, so that the given direction needs to be
converted to  $\theta$  and  $\phi$  values and scaled to [0, 1] before being used to index into the
texture. Goniophotometric diagrams are usually defined in a coordinate space where the
y axis is up, whereas the spherical coordinate utility routines in pbrt assume that z is up,
so y and z are swapped before doing the conversion.

(GonioPhotometricLight Public Methods) +≡ 728
Spectrum Scale(const Vector3f &w) const {
    Vector3f wp = Normalize(WorldToLight(w));
    std::swap(wp.y, wp.z);
    Float theta = SphericalTheta(wp);
    Float phi = SphericalPhi(wp);
    Point2f st(phi * Inv2Pi, theta * InvPi);
    return !mipmap ? RGBSpectrum(1.f) :
        Spectrum(mipmap->Lookup(st), SpectrumType::Illuminant);
}
```

Float 1062
 GonioPhotometricLight 728
 GonioPhotometricLight::I 730
 GonioPhotometricLight::mipmap 730
 GonioPhotometricLight::pLight 730
 Inv2Pi 1063
 InvPi 1063
 Light 714
 Light::LightToWorld 715
 Light::WorldToLocal 715
 LightFlags::DeltaPosition 715
 MediumInterface 684
 MIPMap 625
 MIPMap::Lookup() 635
 Point2f 68
 Point2i 68
 Point3f 68
 ProjectionLight 724
 ProjectionLight::Sample_Li() 727
 ReadImage() 1067
 RGBSpectrum 332
 Spectrum 315
 SpectrumType::Illuminant 330
 SphericalPhi() 346
 SphericalTheta() 346
 SpotLight::Sample_Li() 723
 Transform 83
 Vector3::Normalize() 66
 Vector3f 60

The Power() method uses the average intensity over the image to compute power. This computation is inaccurate because the spherical coordinate parameterization of directions has various distortions, particularly near the $+z$ and $-z$ directions. Again, this error is acceptable for the uses of this method in pbrt.

```
<GonioPhotometricLight Method Definitions> ≡
    Spectrum GonioPhotometricLight::Power() const {
        return 4 * Pi * I *
            Spectrum(mipmap ? mipmap->Lookup(Point2f(.5f, .5f), .5f) :
            Spectrum(1.f), SpectrumType::Illuminant);
    }
```

12.4 DISTANT LIGHTS

Another useful light source type is the *distant light*, also known as a *directional light*. It describes an emitter that deposits illumination from the same direction at every point in space. Such a light is also called a point light “at infinity,” since, as a point light becomes progressively farther away, it acts more and more like a directional light. For example, the sun (as considered from Earth) can be thought of as a directional light source. Although it is actually an area light source, the illumination effectively arrives at Earth in parallel beams because it is so far away. The `DistantLight`, implemented in the files `lights/distant.h` and `lights/distant.cpp`, implements a directional source.

```
<DistantLight Declarations> ≡
class DistantLight : public Light {
public:
    <DistantLight Public Methods 732>
private:
    <DistantLight Private Data 731>
};
```

Note that the `DistantLight` constructor does not take a `MediumInterface` parameter; the only reasonable medium for a distant light to be in is a vacuum—if it was itself in a medium that absorbed any light at all, then all of its emission would be absorbed, since it’s modeled as being infinitely far away.

```
<DistantLight Method Definitions> ≡
DistantLight::DistantLight(const Transform &LightToWorld,
    const Spectrum &L, const Vector3f &wLight)
: Light((int)LightFlags::DeltaDirection, LightToWorld,
    MediumInterface(),
    L(L), wLight(Normalize(LightToWorld(wLight)))) { }
```

```
<DistantLight Private Data> ≡
const Spectrum L;
const Vector3f wLight;
```

DistantLight 731
DistantLight::L 731
DistantLight::wLight 731
GonioPhotometricLight::I 730
GonioPhotometricLight::mipmap 730
Light 714
Light::LightToWorld 715
LightFlags::DeltaDirection 715
MediumInterface 684
MIPMap::Lookup() 635
Pi 1063
Point2f 68
Spectrum 315
SpectrumType::Illuminant 330
Transform 83
Vector3f::Normalize() 66
Vector3f 60

Some of the `DistantLight` methods need to know the bounds of the scene. Because lights are created before the scene geometry, these bounds aren’t available when the `DistantLight` constructor runs. Therefore, `DistantLight` implements the optional `Preprocess()` method to get the bound. (This method is called at the end of the `Scene` constructor.)

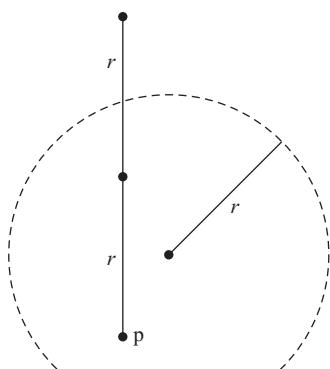


Figure 12.14: Computing the Second Point for a DistantLight Shadow Ray. Given a sphere that bounds the scene (dashed line) with radius r and given some point in the scene p , then if we move a distance of $2r$ along any vector from p , the resulting point must be outside of the scene's bound. If a shadow ray to such a point is unoccluded, then we can be certain that the point p receives illumination from a distant light along the vector's direction.

(DistantLight Public Methods) \equiv

731

```
void Preprocess(const Scene &scene) {
    scene.WorldBound().BoundingSphere(&worldCenter, &worldRadius);
}
```

(DistantLight Private Data) $+ \equiv$

731

```
Point3f worldCenter;
Float worldRadius;
```

Most of the implementation of the `Sample_Li()` method is straightforward: for a distant light, the incident direction and radiance are always the same. The only interesting bit is the initialization of the `VisibilityTester`: here, the second point for the shadow ray is set along the distant light's incident direction a distance of twice the radius of the scene's bounding sphere, guaranteeing a second point that is outside of the scene's bounds (Figure 12.14).

(DistantLight Method Definitions) $+ \equiv$

```
Spectrum DistantLight::Sample_Li(const Interaction &ref,
    const Point2f &u, Vector3f *wi, Float *pdf,
    VisibilityTester *vis) const {
    *wi = wLight;
    *pdf = 1;
    Point3f pOutside = ref.p + wLight * (2 * worldRadius);
    *vis = VisibilityTester(ref, Interaction(pOutside, ref.time,
                                              mediumInterface));
    return L;
}
```

```
Bounds3::BoundingSphere() 81
DistantLight 731
DistantLight::L 731
DistantLight::wLight 731
DistantLight::worldCenter 732
DistantLight::worldRadius 732
Float 1062
Interaction 115
Interaction::p 115
Interaction::time 115
Light::mediumInterface 715
Point2f 68
Point3f 68
Scene 23
Scene::WorldBound() 24
Spectrum 315
Vector3f 60
VisibilityTester 717
```

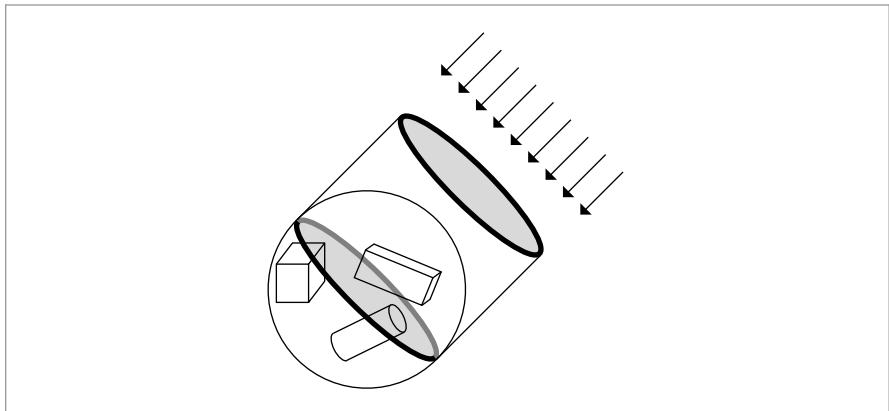


Figure 12.15: An approximation of the power emitted by a distant light into a given scene can be obtained by finding the sphere that bounds the scene, computing the area of an inscribed disk, and computing the power that arrives on the surface of that disk.

The distant light is unusual in that the amount of power it emits is related to the spatial extent of the scene. In fact, it is proportional to the area of the scene receiving light. To see why this is so, consider a disk of area A being illuminated by a distant light with emitted radiance L where the incident light arrives along the disk's normal direction. The total power reaching the disk is $\Phi = AL$. As the size of the receiving surface varies, power varies proportionally.

To find the emitted power for a `DistantLight`, it's impractical to compute the total surface area of the objects that are visible to the light. Instead, we will approximate this area with a disk inside the scene's bounding sphere oriented in the light's direction (Figure 12.15). This will always overestimate the actual area but is sufficient for the needs of code elsewhere in the system.

```
(DistantLight Method Definitions) +≡
Spectrum DistantLight::Power() const {
    return L * Pi * worldRadius * worldRadius;
}
```

12.5 AREA LIGHTS

`DistantLight` 731
`DistantLight::L` 731
`DistantLight::worldRadius` 732
`Pi` 1063
`Shape` 123
`Spectrum` 315

Area lights are light sources defined by one or more `Shapes` that emit light from their surface, with some directional distribution of radiance at each point on the surface. In general, computing radiometric quantities related to area lights requires computing integrals over the surface of the light that often can't be computed in closed form. This issue is addressed with the Monte Carlo integration techniques in Section 14.2. The reward for this complexity (and computational expense) is soft shadows and more realistic lighting effects, rather than the hard shadows and stark lighting that come from point lights. (See Figure 12.16. Also compare Figure 12.17 to Figure 12.6.)



Figure 12.16: Wider View of the Lighting Example Scene. The dragon is illuminated by a disk area light source directly above it.

The `AreaLight` class is an abstract base class that inherits from `Light`. Implementations of area lights should inherit from it.

(Light Declarations) +≡

```
class AreaLight : public Light {
public:
    (AreaLight Interface 734)
};
```

`AreaLight` adds a single new method to the general `Light` interface, `AreaLight::L()`. Implementations are given a point on the surface of the light represented by an `Interaction` and should evaluate the area light's emitted radiance, L , in the given outgoing direction.

(AreaLight Interface) ≡

```
virtual Spectrum L(const Interaction &intr, const Vector3f &w) const = 0;
```

For convenience, there is a method in the `SurfaceInteraction` class that makes it easy to compute the emitted radiance at a surface point intersected by a ray.

(SurfaceInteraction Method Definitions) +≡

```
Spectrum SurfaceInteraction::Le(const Vector3f &w) const {
    const AreaLight *area = primitive->GetAreaLight();
    return area ? area->L(*this, w) : Spectrum(0.f);
}
```

`DiffuseAreaLight` implements a basic area light source with a uniform spatial and directional radiance distribution. The surface it emits from is defined by a `Shape`. It only emits

`AreaLight` [734](#)
`AreaLight::L()` [734](#)
`DiffuseAreaLight` [736](#)
`Interaction` [115](#)
`Light` [714](#)
`Primitive::GetAreaLight()` [249](#)
`Shape` [123](#)
`Spectrum` [315](#)
`SurfaceInteraction` [116](#)
`Vector3f` [60](#)



(a)



(b)

Figure 12.17: Dragon Model Illuminated by Disk Area Lights. (a) The disk's radius is relatively small; the shadow has soft penumbrae, but otherwise the image looks similar to the one with a point light. (b) The effect of using a much larger disk: not only have the penumbrae become much larger, to the point of nearly eliminating the fully in-shadow areas, but notice how areas like the neck of the dragon and its jaw have noticeably different appearances when illuminated from a wider range of directions.

light on the side of the surface with outward-facing surface normal; there is no emission from the other side. (The `Shape::reverseOrientation` value can be set to true to cause the light to be emitted from the other side of the surface instead.) `DiffuseAreaLight` is defined in the files `lights/diffuse.h` and `lights/diffuse.cpp`.

(DiffuseAreaLight Declarations) ≡

```
class DiffuseAreaLight : public AreaLight {
public:
    (DiffuseAreaLight Public Methods 736)
protected:
    (DiffuseAreaLight Protected Data 736)
};
```

(DiffuseAreaLight Method Definitions) ≡

```
DiffuseAreaLight::DiffuseAreaLight(const Transform &LightToWorld,
    const MediumInterface &mediumInterface, const Spectrum &Lemit,
    int nSamples, const std::shared_ptr<Shape> &shape)
: AreaLight(LightToWorld, mediumInterface, nSamples), Lemit(Lemit),
  shape(shape), area(shape->Area()) { }
```

(DiffuseAreaLight Protected Data) ≡

736

```
const Spectrum Lemit;
std::shared_ptr<Shape> shape;
const Float area;
```

Because this area light implementation emits light from only one side of the shape's surface, its `L()` method just makes sure that the outgoing direction lies in the same hemisphere as the normal.

(DiffuseAreaLight Public Methods) ≡

736

```
Spectrum L(const Interaction &intr, const Vector3f &w) const {
    return Dot(intr.n, w) > 0.f ? Lemit : Spectrum(0.f);
}
```

The `DiffuseAreaLight::Sample_Li()` method isn't as straightforward as it has been for the other light sources described so far. Specifically, at each point in the scene, radiance from area lights can be incident from many directions, not just a single direction as was the case for the other lights (Figure 12.18). This leads to the question of which direction should be chosen for this method. We will defer answering this question and providing an implementation of this method until Section 14.2, after Monte Carlo integration has been introduced.

Emitted power from an area light with uniform emitted radiance over the surface can be directly computed in closed form:

(DiffuseAreaLight Method Definitions) +≡

```
Spectrum DiffuseAreaLight::Power() const {
    return Lemit * area * Pi;
}
```

AreaLight 734
 DiffuseAreaLight 736
`DiffuseAreaLight::area` 736
`DiffuseAreaLight::Lemit` 736
`DiffuseAreaLight::Sample_Li()` 845
`DiffuseAreaLight::shape` 736
`Dot()` 63
`Float` 1062
`Interaction` 115
`Interaction::n` 116
`MediumInterface` 684
`Pi` 1063
`Shape` 123
`Shape::Area()` 131
`Shape::reverseOrientation` 124
`Spectrum` 315
`Transform` 83
`Vector3f` 60

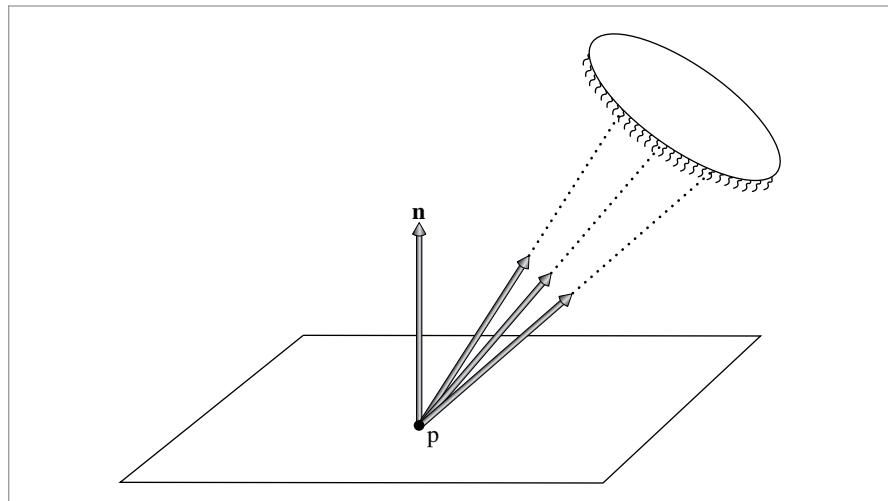


Figure 12.18: An area light casts incident illumination along many directions, rather than from a single direction.

12.6 INFINITE AREA LIGHTS

Another useful kind of light is the infinite area light—an infinitely far away area light source that surrounds the entire scene. One way to visualize this light is as an enormous sphere that casts light into the scene from every direction. One important use of infinite area lights is for *environment lighting*, where an image that represents illumination in an environment is used to illuminate synthetic objects as if they were in that environment. Figures 12.19 and 12.20 compare illuminating a car model with a standard area light to illuminating it with environment maps that simulate illumination from the sky at a few different times of day (the illumination maps used are shown in Figure 12.21). The increase in realism is striking. The `InfiniteAreaLight` class is implemented in `lights/infinite.h` and `lights/infinite.cpp`.

A widely used representation for light for this application is the latitude-longitude radiance map. (This representation is also known as the equirectangular projection.) The `EnvironmentCamera` can be used to create image maps for the light, or see the “Further Reading” section for information about techniques for capturing this lighting data from real-world environments.

(InfiniteAreaLight Declarations) ≡

```
class InfiniteAreaLight : public Light {
public:
    (InfiniteAreaLight Public Methods 740)
private:
    (InfiniteAreaLight Private Data 740)
};
```

`EnvironmentCamera` 376

`InfiniteAreaLight` 737

`Light` 714



(a)



(b)

Figure 12.19: Car model (a) illuminated with an area light and a directional light and (b) illuminated with morning skylight from an environment map. Using a realistic distribution of illumination gives an image that is much more visually compelling. In particular, with illumination arriving from all directions, the glossy reflective properties of the paint are much more visually apparent. (*Model courtesy of Yasutoshi Mori.*)

Like the other lights, the `InfiniteAreaLight` takes a transformation matrix; here, its use is to orient the image map. It then uses spherical coordinates to map from directions on the sphere to (θ, ϕ) directions, and from there to (u, v) texture coordinates. The provided transformation thus determines which direction is “up.”

The constructor loads the image data from the disk and creates a `MIPMap` to store it. The fragment that loads the data, *(Read texel data from texmap and initialize Lmap)*, is straightforward and won’t be included here. The other code fragment in the constructor, *(Initialize sampling PDFs for infinite area light)*, is related to Monte Carlo sampling of `InfiniteAreaLights` and will be defined later, in Section 14.2.4.

`InfiniteAreaLight` 737

`MIPMap` 625



(a)



(b)

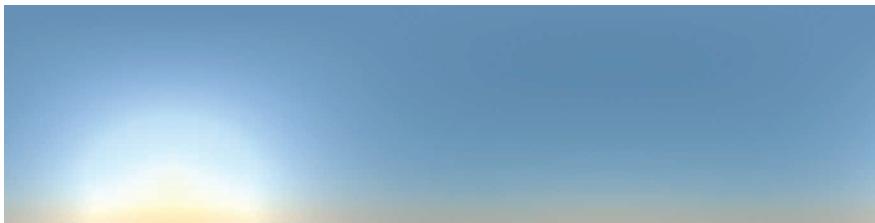
Figure 12.20: Changing just the environment map used for illumination gives quite different results in the final image: (a) using a midday skylight distribution and (b) using a sunset environment map. (Model courtesy of Yasutoshi Mori.)

As with `DistantLight`, because the light is defined as being infinitely far away, the `MediumInterface` for an infinite area light must have `nullptr` values for its `Medium`'s, corresponding to a vacuum.

`DistantLight` 731
`InfiniteAreaLight` 737
`Light` 714
`LightFlags::Infinite` 715
`MediumInterface` 684
`Spectrum` 315
`Transform` 83

```

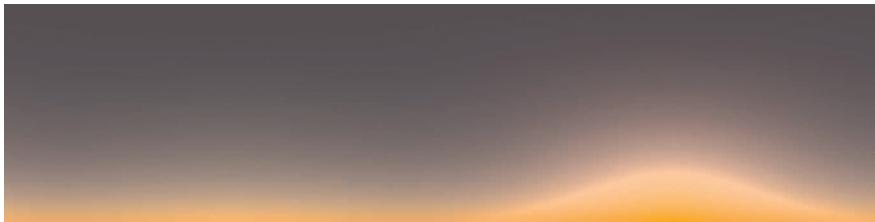
<InfiniteAreaLight Method Definitions> ≡
InfiniteAreaLight::InfiniteAreaLight(const Transform &LightToWorld,
                                     const Spectrum &L, int nSamples, const std::string &texmap)
: Light((int)LightFlags::Infinite, LightToWorld,
        MediumInterface(), nSamples) {
  (Read texel data from texmap and initialize Lmap)
  (Initialize sampling PDFs for infinite area light 847)
}
  
```



(a)



(b)



(c)

Figure 12.21: Environment Maps Used for Illumination in Figures 12.19 and 12.20. (a) Morning, (b) midday, and (c) sunset sky. (The bottom halves of these maps aren't shown here, since they are just black pixels.)

(InfiniteAreaLight Private Data) ≡

737

```
std::unique_ptr<MIPMap<RGBSpectrum>> Lmap;
```

Like DistantLights, InfiniteAreaLights also need the scene bounds; here again, the Preprocess() method finds the scene bounds after all of the scene geometry has been created.

(InfiniteAreaLight Public Methods) ≡

737

```
void Preprocess(const Scene &scene) {
    scene.WorldBound().BoundingSphere(&worldCenter, &worldRadius);
}
```

(InfiniteAreaLight Private Data) +≡

737

```
Point3f worldCenter;
Float worldRadius;
```

Bounds3::BoundingSphere() 81

DistantLight 731

Float 1062

InfiniteAreaLight::
worldCenter
740

InfiniteAreaLight::
worldRadius
740

MIPMap 625

Point3f 68

RGBSpectrum 332

Scene 23

Scene::WorldBound() 24

Because `InfiniteAreaLights` cast light from all directions, it's also necessary to use Monte Carlo integration to sample their illumination. Therefore, the `InfiniteAreaLight::Sample_Li()` method will be defined in Section 14.2.

Like directional lights, the total power from the infinite area light is related to the surface area of the scene. Like many other lights in this chapter, the power computed here is approximate; here, all texels are given equal weight, which ignores the fact that with an equirectangular projection, the differential solid angle subtended by each pixel values with its θ value (Section 14.2.4).

(InfiniteAreaLight Method Definitions) +≡

```
Spectrum InfiniteAreaLight::Power() const {
    return Pi * worldRadius * worldRadius *
        Spectrum(Lmap->Lookup(Point2f(.5f, .5f), .5f),
                  SpectrumType::Illuminant);
}
```

Because infinite area lights need to be able to contribute radiance to rays that don't hit any geometry in the scene, we'll add a method to the base `Light` class that returns emitted radiance due to that light along a ray that escapes the scene bounds. (The default implementation for other lights returns no radiance.) It is the responsibility of the integrators to call this method for these rays.

(Light Method Definitions) +≡

```
Spectrum Light::Le(const RayDifferential &ray) const {
    return Spectrum(0.f);
}
```

`InfiniteAreaLight` 737

`InfiniteAreaLight::Lmap` 740

`InfiniteAreaLight::`
 `Sample_Li()`
 849

`InfiniteAreaLight::`
 `worldRadius`
 740

`InvPi` 1063

`InvPi` 1063

`Light` 714

`Light::WorldToLight` 715

`MIPMap::Lookup()` 635

`Pi` 1063

`Point2f` 68

`Ray::d` 73

`RayDifferential` 75

`Spectrum` 315

`SpectrumType::Illuminant` 330

`SphericalPhi()` 346

`SphericalTheta()` 346

`Vector3f::Normalize()` 66

`Vector3f` 60

(InfiniteAreaLight Method Definitions) +≡

```
Spectrum InfiniteAreaLight::Le(const RayDifferential &ray) const {
    Vector3f w = Normalize(WorldToLight(ray.d));
    Point2f st(SphericalPhi(w) * Inv2Pi,
                SphericalTheta(w) * InvPi);
    return Spectrum(Lmap->Lookup(st), SpectrumType::Illuminant);
}
```

FURTHER READING

The books by McCluney (1994) and Malacara (2002) discuss blackbody emitters and the standard illuminants in detail. Wilkie and Weidlich (2011) noted that common practice in rendering has been to use the blackbody distribution of Equation (12.1) to model light emission for rendering, while Kirchoff's law, Equation (12.4), would be more accurate. They also point out that as objects become hot, their BRDFs often change, which makes Kirchoff's law more difficult to adopt, especially in that models that account for the effect of temperature variation on BRDFs generally aren't available.

The Standard Illuminants are defined in a CIE Technical Report (2004); Judd et al. (1964) developed the approach that was used to define the D Standard Illuminant.

Warn (1983) developed early models of light sources with nonisotropic emission distributions, including the spotlight model used in this chapter. Verbeck and Greenberg (1984) also described a number of techniques for modeling light sources that are now classic parts of the light modeling toolbox. Barzel (1997) described a highly parameterized model for light sources, including many controls for controlling rate of falloff, the area of space that is illuminated, and so on. Bjorke (2001) described a number of additional controls for controlling illumination for artistic effect. (Many parts of the Barzel and Bjorke approaches are not physically based, however.)

The goniometric light source approximation is widely used to model area light sources in the field of illumination engineering. The rule of thumb there is that once a reference point is five times an area light source's radius away from it, a point light approximation has sufficient accuracy for most applications. File format standards have been developed for encoding goniophotometric diagrams for these applications (Illuminating Engineering Society of North America 2002). Many lighting fixture manufacturers provide data in these formats on their Web sites.

Ashdown (1993) proposed a more sophisticated light source model than goniometric diagrams; he measured the directional distribution of emitted radiance at a large number of points around a light source and described how to use the resulting 4D table to compute the received radiance distribution at other points. Another generalization of goniometric lights was suggested by Heidrich et al. (1998), who represented light sources as a 4D exitant *lightfield*—essentially a function of both position and direction—and showed how to use this representation for rendering. Additional work in this area was done by Goesele et al. (2003), who further investigated issues in measuring light sources, and Mas et al. (2008), who introduced a more space-efficient representation and improved rendering efficiency.

Real-world light sources are often fairly complex, including carefully designed systems of mirrors and lenses to shape the distribution of light emitted by the light source. (Consider, for example, the headlights on a car, where it's important to evenly illuminate the surface of the road, without shining too much light in the eyes of approaching drivers.) As we'll see in Chapter 16, all of this specular reflection and transmission is challenging for light transport algorithms. It can therefore be worthwhile to do some precomputation to create a representation of light sources' final emission distributions after all of this scattering that is then used as the light source model for rendering. To this end, Kniep et al. (2009) suggest tracing the paths of photons leaving the light's filament until they hit a bounding surface around the light. They then record the position and direction of outgoing photons and use this information when computing illumination at points in the scene. Velázquez-Armendáriz et al. (2015) showed how to compute a set of point lights with directionally varying emission distributions to model emitted radiance from complex light sources. They then approximated the radiance distribution in the light interior using spherical harmonics.

Blinn and Newell (1976) first introduced the idea of environment maps and their use for simulating illumination, although they only considered illumination of specular objects. Greene (1986) further refined these ideas, considering antialiasing and different representations for environment maps. Nishita and Nakamae (1986) developed algorithms for

efficiently rendering objects illuminated by hemispherical skylights and generated some of the first images that showed off that distinctive lighting effect. Miller and Hoffman (1984) were the first to consider using arbitrary environment maps to illuminate objects with diffuse and glossy BRDFs. Debevec (1998) later extended this work and investigated issues related to capturing images of real environments.

Representing illumination from the sun and sky is a particularly important application of infinite light sources; the “Further Reading” section in Chapter 15 includes a number of references related to simulating skylight scattering. Directly measuring illumination the sky is also an effective way to find accurate skylight illumination; see Kider et al. (2014) for details of a system they built to do this. (A companion Web site has a large amount of measured skylight data available for download.)

pbrt’s infinite area light source models incident radiance from the light as purely a function of direction. Especially for indoor scenes, this assumption can be fairly inaccurate; position matters as well. Unger et al. (2003) captured the incident radiance as a function of direction at many different locations in a real-world scene and used this representation for rendering. Unger et al. (2008) improved on this work and showed how to decimate the samples to reduce storage requirements without introducing too much error.

As discussed in Chapter 3, one way to reduce the time spent tracing shadow rays is to have methods like `Shape::IntersectP()` and `Primitive::IntersectP()` that just check for any occlusion along a ray without bothering to compute the geometric information at the intersection point. Other approaches for optimizing ray tracing for shadow rays include the *shadow cache*, where each light stores a pointer to the last primitive that occluded a shadow ray to the light. That primitive is checked first to see if it occludes subsequent shadow rays before the ray is passed to the acceleration structure (Haines and Greenberg 1986). Pearce (1991) pointed out that the shadow cache doesn’t work well if the scene has finely tessellated geometry; it may be better to cache the BVH node that held the last occluder, for instance. (The shadow cache can similarly be defeated when multiple levels of reflection and refraction are present or when Monte Carlo ray-tracing techniques are used.) Hart, Dutré, and Greenberg (1999) developed a generalization of the shadow cache, which tracks which objects block light from particular light sources and clips their geometry against the light source geometry so that shadow rays don’t need to be traced toward the parts of the light that are certain to be occluded.

A related technique, described by Haines and Greenberg (1986), is the light buffer for point light sources, where the light discretizes the directions around it and determines which objects are visible along each set of directions (and are thus potential occluding objects for shadow rays). Another effective optimization is *shaft culling*, which takes advantage of coherence among groups of rays traced in a similar set of directions (e.g., shadow rays from a single point to points on an area light source). With shaft culling, a shaft that bounds a collection of rays is computed and then the objects in the scene that penetrate the shaft are found. For all of the rays in the shaft, it is only necessary to check for intersections with those objects that intersect the shaft, and the expense of ray intersection acceleration structure traversal for each of the rays is avoided (Haines and Wallace 1994).

`Primitive::IntersectP()` 249

`Shape::IntersectP()` 130

Woo and Amanatides (1990) classified which lights are visible, not visible, and partially visible in different parts of the scene and stored this information in a voxel-based 3D data structure, using this information to save shadow ray tests. Fernandez, Bala, and Greenberg (2002) developed a similar approach based on spatial decomposition that stores references to important blockers in each voxel and also builds up this information on demand for applications like walkthroughs.

For complex models, simplified versions of their geometry can be used for shadow ray intersections. For example, the simplification envelopes described by Cohen et al. (1996) can create a simplified mesh that bounds a given mesh from both the inside and the outside. If a ray misses the mesh that bounds a complex model from the outside or intersects the mesh that bounds it from the inside, then no further shadow processing is necessary. Only the uncertain remaining cases need to be intersected against the full geometry. A related technique is described by Lukaszewski (2001), who uses the Minkowski sum to effectively expand primitives (or bounds of primitives) in the scene so that intersecting one ray against one of these primitives can determine if any of a collection of rays might have intersected the actual primitives.

EXERCISES

- ② 12.1 Shadow mapping is a technique for rendering shadows from point and distant light sources based on rendering an image from the light source's perspective that records depth in each pixel of the image and then projecting points onto the shadow map and comparing their depth to the depth of the first visible object as seen from the light in that direction. This method was first described by Williams (1978), and Reeves, Salesin, and Cook (1987) developed a number of key improvements. Modify pbrt to be able to render depth map images into a file and then use them for shadow testing for lights in place of tracing shadow rays. How much faster can this be? Discuss the advantages and disadvantages of the two approaches.
- ① 12.2 Through algebraic manipulation and precomputation of one more value in the constructor, the `SpotLight::FallOff()` method can be rewritten to compute the exact same result (modulo floating-point differences) while using no square root computations and no divides (recall that the `Vector3::Normalize()` method performs both a square root and a divide). Derive and implement this optimization. How much is running time improved on a spotlight-heavy scene?
- ① 12.3 The functionality of the `SpotLight` could be replicated by using a suitable image in conjunction with the `ProjectionLight`. Discuss the advantages and disadvantages of providing this specific functionality separately with the `SpotLight` class.
- ③ 12.4 The current light source implementations don't support animated transformations. Modify pbrt to include this functionality, and render images showing off the effect of animating light positions.

`SpotLight::FallOff()` 724

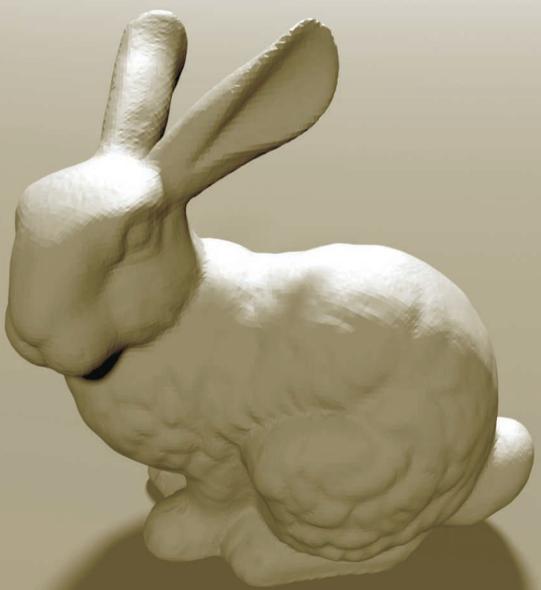
`Vector3::Normalize()` 66

- ② 12.5 Modify the `ProjectionLight` to also support orthographic projections. This variant is particularly useful even without an image map, since it gives a directional light source with a beam of user-defined extent.
- ② 12.6 Write an `AreaLight` implementation that improves on the `DiffuseAreaLight` by supporting spatially and directionally varying emitted radiance, specified via either image maps or `Textures`. Use it to render images with effects like a television illuminating a dark room or a stained-glass window lit from behind.
- ② 12.7 Many of the `Light::Power()` method implementations only compute approximations to the actual emitted power for their lights. In particular, all of the lights that use images (`ProjectionLight`, `GonioPhotometricLight`, and `InfiniteAreaLight`) all neglect the fact that for each of them, different pixels subtend different solid angles and therefore contribute differently to the emitted power. Derive accurate models for the emitted power of these light sources, and implement them in `pbrt`. How much error do the current implementations have when used in some of the `pbrt` example scenes? Can you construct contrived scenes to show the maximum error introduced by the current implementation?
- ② 12.8 Read some of the papers in the “Further Reading” section that discuss the shadow cache, and add this optimization to `pbrt`. Measure how much it speeds up the system for a variety of scenes. What techniques can you come up with that make it work better in the presence of multiple levels of reflection?
- ③ 12.9 Modify `pbrt` to support the shaft culling algorithm (Haines and Wallace 1994). Measure the performance difference for scenes with area light sources. Make sure that your implementation still performs well even with very large light sources (like a hemispherical skylight).
- ③ 12.10 Read the paper by Velázquez-Armendáriz et al. (2015), and implement their method for efficiently rendering scenes with complex light sources. Create or find models of a few complex lights, including many shapes that exhibit specular reflection and/or transmission. Compare results using your implementation to renderings using one or more of the bidirectional integrators from Chapter 16 (which are best suited to handling this challenge). Note that you may need to set very long maximum integrator path lengths for the current implementation of `pbrt` to be able to render these scenes at all.

How much more efficiently does your implementation render images of scenes lit by these lights than the built-in integrators? Do results from the two approaches match?

`AreaLight` 734
`DiffuseAreaLight` 736
`GonioPhotometricLight` 728
`InfiniteAreaLight` 737
`Light::Power()` 717
`ProjectionLight` 724
`Texture` 614

CHAPTER THIRTEEN



13 MONTE CARLO INTEGRATION

Before we introduce the Integrators that compute radiance along ray paths between lights and the camera, we will first lay some groundwork regarding the techniques they will use to compute solutions to the integral equations that describe light scattering. These integral equations generally do not have analytic solutions, so we must turn to numerical methods. Although standard numerical integration techniques like trapezoidal integration or Gaussian quadrature are very effective at solving low-dimensional smooth integrals, their rate of convergence for the higher dimensional and discontinuous integrals that are common in rendering is poor.

Monte Carlo numerical integration methods provide one solution to this problem. They use randomness to evaluate integrals with a convergence rate that is independent of the dimensionality of the integrand. In this chapter, we review important concepts from probability and lay the foundation for using Monte Carlo techniques to evaluate the key integrals in rendering.

Judicious use of randomness has revolutionized the field of algorithm design. Randomized algorithms fall broadly into two classes: *Las Vegas* and *Monte Carlo*. Las Vegas algorithms are those that use randomness but always give the same result in the end (e.g., choosing a random array entry as the pivot element in Quicksort). Monte Carlo algorithms, on the other hand, give different results depending on the particular random numbers used along the way but give the right answer *on average*. So, by averaging the results of several runs of a Monte Carlo algorithm (on the same input), it is possible to find a result that is statistically very likely to be close to the true answer. Motwani and Raghavan (1995) have written an excellent introduction to the field of randomized algorithms.

Monte Carlo integration¹ is a method for using random sampling to estimate the values of integrals. One very useful property of Monte Carlo is that one only needs the ability to evaluate an integrand $f(x)$ at arbitrary points in the domain in order to estimate the value of its integral $\int f(x) dx$. This property not only makes Monte Carlo easy to implement but also makes the technique applicable to a broad variety of integrands, including those containing discontinuities.

Many of the integrals that arise in rendering are difficult or impossible to evaluate directly. For example, to compute the amount of light reflected by a surface at a point according to Equation (5.9), we must integrate the product of the incident radiance and the BSDF over the unit sphere. How to do so is not immediately clear: the incident radiance function is almost never available in closed form due to the complex and difficult-to-predict effect of object visibility in realistic scenes.

Even if the incident radiance function were available in closed form, performing the integral analytically would still not be possible in general. Monte Carlo integration makes it possible to estimate the reflected radiance simply by choosing a set of directions over the sphere, computing the incident radiance along them, multiplying by the BSDF's value for those directions, and applying a weighting term. Arbitrary BSDFs, light source descriptions, and scene geometry are easily handled; evaluation of each of these functions at arbitrary points is all that is required.

The main disadvantage of Monte Carlo is that if n samples are used to estimate the integral, the algorithm converges to the correct result at a rate of $O(n^{-1/2})$. In other words, to cut the error in half, it is necessary to evaluate four times as many samples. In rendering, each sample generally requires that one or more rays be traced in the process of computing the value of the integrand, a computationally expensive cost to bear when using Monte Carlo for image synthesis. In images, artifacts from Monte Carlo sampling manifest themselves as noise—pixels are randomly too bright or too dark. Most of the current research in Monte Carlo for computer graphics is about reducing this error as much as possible while minimizing the number of additional samples that must be taken.

13.1 BACKGROUND AND PROBABILITY REVIEW

We will start by defining some terms and reviewing basic ideas from probability. We assume that the reader is already familiar with basic probability concepts; readers needing a more complete introduction to this topic should consult a textbook such as Sheldon Ross's *Introduction to Probability Models* (2002).

A *random variable* X is a value chosen by some random process. We will generally use capital letters to denote random variables, with exceptions made for a few Greek symbols that represent special random variables. Random variables are always drawn from some domain, which can be either discrete (e.g., a fixed set of possibilities) or continuous (e.g., the real numbers \mathbb{R}). Applying a function f to a random variable X results in a new random variable $Y = f(X)$.

¹ For brevity, we will refer to Monte Carlo integration simply as "Monte Carlo."

For example, the result of a roll of a die is a discrete random variable sampled from the set of events $X_i = \{1, 2, 3, 4, 5, 6\}$. Each event has a probability $p_i = \frac{1}{6}$, and the sum of probabilities $\sum p_i$ is necessarily one. We can take a continuous, uniformly distributed random variable $\xi \in [0, 1)$ and map it to a discrete random variable, choosing X_i if

$$\sum_{j=1}^{i-1} p_j < \xi \leq \sum_{j=1}^i p_j.$$

For lighting applications, we might want to define the probability of sampling illumination from each light in the scene based on the power Φ_i from each source relative to the total power from all sources:

$$p_i = \frac{\Phi_i}{\sum_j \Phi_j}.$$

Notice that these p_i also sum to 1.

The *cumulative distribution function* (CDF) $P(x)$ of a random variable is the probability that a value from the variable's distribution is less than or equal to some value x :

$$P(x) = \Pr\{X \leq x\}.$$

For the die example, $P(2) = \frac{1}{3}$, since two of the six possibilities are less than or equal to 2.

13.1.1 CONTINUOUS RANDOM VARIABLES

In rendering, discrete random variables are less common than continuous random variables, which take on values over ranges of continuous domains (e.g., the real numbers, directions on the unit sphere, or the surfaces of shapes in the scene).

A particularly important random variable is the *canonical uniform random variable*, which we will write as ξ . This variable takes on all values in its domain $[0, 1)$ with equal probability. This particular variable is important for two reasons. First, it is easy to generate a variable with this distribution in software—most run-time libraries have a pseudo-random number generator that does just that.² Second, as we will show later, it is possible to generate samples from arbitrary distributions by first starting with canonical uniform random variables and applying an appropriate transformation. The technique described previously for mapping from ξ to the six faces of a die gives a flavor of this technique in the discrete case.

Another example of a continuous random variable is one that ranges over the real numbers between 0 and 2, where the probability of its taking on any particular value x is proportional to the value $2 - x$: it is twice as likely for this random variable to take on a value around 0 as it is to take one around 1, and so forth. The *probability density function* (PDF) formalizes this idea: it describes the relative probability of a random variable

² Although the theory of Monte Carlo is based on using truly random numbers, in practice a well-written pseudo-random number generator (PRNG) is sufficient. pbrt uses a particularly high-quality PRNG that returns a sequence of pseudo-random values that is effectively as “random” as true random numbers. (Many PRNGs are not as well implemented and have detectable patterns in the sequence of numbers they generate.) True random numbers, found by measuring random phenomena like atomic decay or atmospheric noise, are available from sources like www.random.org for those for whom PRNGs are not acceptable.

taking on a particular value. The PDF $p(x)$ is the derivative of the random variable's CDF,

$$p(x) = \frac{dP(x)}{dx}.$$

For uniform random variables, $p(x)$ is a constant; this is a direct consequence of uniformity. For ξ we have

$$p(x) = \begin{cases} 1 & x \in [0, 1] \\ 0 & \text{otherwise.} \end{cases}$$

PDFs are necessarily nonnegative and always integrate to 1 over their domains. Given an arbitrary interval $[a, b]$ in the domain, integrating the PDF gives the probability that a random variable lies inside the interval:

$$P(x \in [a, b]) = \int_a^b p(x) dx.$$

This follows directly from the first fundamental theorem of calculus and the definition of the PDF.

13.1.2 EXPECTED VALUES AND VARIANCE

The *expected value* $E_p[f(x)]$ of a function f is defined as the average value of the function over some distribution of values $p(x)$ over its domain. In the next section, we will see how Monte Carlo integration computes the expected values of arbitrary integrals. The expected value over a domain, D , is defined as

$$E_p[f(x)] = \int_D f(x) p(x) dx. \quad [13.1]$$

As an example, consider the problem of finding the expected value of the cosine function between 0 and π , where p is uniform.³ Because the PDF $p(x)$ must integrate to 1 over the domain, $p(x) = 1/\pi$, so

$$E[\cos x] = \int_0^\pi \frac{\cos x}{\pi} dx = \frac{1}{\pi}(-\sin \pi + \sin 0) = 0,$$

which is precisely the expected result. (Consider the graph of $\cos x$ over $[0, \pi]$ to see why this is so.)

The *variance* of a function is the expected squared deviation of the function from its expected value. Variance is a fundamental concept for quantifying the error in a value estimated by a Monte Carlo algorithm. It provides a precise way to quantify this error and measure how improvements to Monte Carlo algorithms reduce the error in the final result. The variance of a function f is defined as

$$V[f(x)] = E[(f(x) - E[f(x)])^2].$$

³ When computing expected values with a uniform distribution, we will drop the subscript p from E_p .

The expected value and variance have a few important properties that follow immediately from their respective definitions:

$$\begin{aligned} E[af(x)] &= aE[f(x)] \\ E\left[\sum_i f(X_i)\right] &= \sum_i E[f(X_i)] \\ V[af(x)] &= a^2V[f(x)]. \end{aligned}$$

These properties, and some simple algebraic manipulation, yield an alternative expanded expression for the variance:

$$V[f(x)] = E\left[(f(x))^2\right] - E[f(x)]^2. \quad (13.2)$$

Thus, the variance is the expected value of the square minus the square of the expected value. Given random variables that are *independent*, variance also has the property that the sum of the variances is equal to the variance of their sum:

$$\sum_i V[f(X_i)] = V\left[\sum_i f(X_i)\right].$$

13.2 THE MONTE CARLO ESTIMATOR

We can now define the basic Monte Carlo estimator, which approximates the value of an arbitrary integral. It is the foundation of the light transport algorithms defined in Chapters 14, 15, and 16.

Suppose that we want to evaluate a 1D integral $\int_a^b f(x) dx$. Given a supply of uniform random variables $X_i \in [a, b]$, the Monte Carlo estimator says that the expected value of the estimator

$$F_N = \frac{b-a}{N} \sum_{i=1}^N f(X_i),$$

$E[F_N]$, is in fact equal to the integral.⁴ This fact can be demonstrated with just a few steps. First, note that the PDF $p(x)$ corresponding to the random variable X_i must be equal to $1/(b-a)$, since p must both be a constant and also integrate to 1 over the domain $[a, b]$. Algebraic manipulation then shows that

Lerp() 1079

RNG::UniformFloat() 1066

4 For example, the samples X_i might be computed in an implementation by Lerp(rng.UniformFloat(), a, b).

$$\begin{aligned}
E[F_N] &= E \left[\frac{b-a}{N} \sum_{i=1}^N f(X_i) \right] \\
&= \frac{b-a}{N} \sum_{i=1}^N E[f(X_i)] \\
&= \frac{b-a}{N} \sum_{i=1}^N \int_a^b f(x) p(x) dx \\
&= \frac{1}{N} \sum_{i=1}^N \int_a^b f(x) dx \\
&= \int_a^b f(x) dx.
\end{aligned}$$

The restriction to uniform random variables can be relaxed with a small generalization. This is an extremely important step, since carefully choosing the PDF from which samples are drawn is an important technique for reducing variance in Monte Carlo (Section 13.10). If the random variables X_i are drawn from some arbitrary PDF $p(x)$, then the estimator

$$F_N = \frac{1}{N} \sum_{i=1}^N \frac{f(X_i)}{p(X_i)} \quad (13.3)$$

can be used to estimate the integral instead. The only limitation on $p(x)$ is that it must be nonzero for all x where $|f(x)| > 0$. It is similarly not too hard to see that the expected value of this estimator is the desired integral of f :

$$\begin{aligned}
E[F_N] &= E \left[\frac{1}{N} \sum_{i=1}^N \frac{f(X_i)}{p(X_i)} \right] \\
&= \frac{1}{N} \sum_{i=1}^N \int_a^b \frac{f(x)}{p(x)} p(x) dx \\
&= \frac{1}{N} \sum_{i=1}^N \int_a^b f(x) dx \\
&= \int_a^b f(x) dx.
\end{aligned}$$

Extending this estimator to multiple dimensions or complex integration domains is straightforward. N samples X_i are taken from a multidimensional (or “joint”) PDF, and the estimator is applied as usual. For example, consider the 3D integral

$$\int_{x_0}^{x_1} \int_{y_0}^{y_1} \int_{z_0}^{z_1} f(x, y, z) dx dy dz.$$

If samples $X_i = (x_i, y_i, z_i)$ are chosen uniformly from the box from (x_0, y_0, z_0) to (x_1, y_1, z_1) , the PDF $p(X)$ is the constant value

$$\frac{1}{(x_1 - x_0)} \frac{1}{(y_1 - y_0)} \frac{1}{(z_1 - z_0)},$$

and the estimator is

$$\frac{(x_1 - x_0)(y_1 - y_0)(z_1 - z_0)}{N} \sum_i f(X_i).$$

Note that the number of samples N can be chosen arbitrarily, regardless of the dimension of the integrand. This is another important advantage of Monte Carlo over traditional deterministic quadrature techniques. The number of samples taken in Monte Carlo is completely independent of the dimensionality of the integral, while with standard numerical quadrature techniques the number of samples required is exponential in the dimension.

Showing that the Monte Carlo estimator converges to the right answer is not enough to justify its use; a good rate of convergence is important too. Although we will not derive its rate of convergence here, it has been shown that error in the Monte Carlo estimator decreases at a rate of $O(\sqrt{N})$ in the number of samples taken. An accessible treatment of this topic can be found in Veach's thesis (Veach 1997, p. 39). Although standard quadrature techniques converge faster than $O(\sqrt{N})$ in one dimension, their performance becomes exponentially worse as the dimensionality of the integrand increases, while Monte Carlo's convergence rate is independent of the dimension, making Monte Carlo the only practical numerical integration algorithm for high-dimensional integrals. We have already encountered some high-dimensional integrals in this book, and in Chapter 14 we will see that the path tracing formulation of the light transport equation is an *infinite-dimensional* integral!

13.3 SAMPLING RANDOM VARIABLES

In order to evaluate the Monte Carlo estimator in Equation (13.3), it is necessary to be able to draw random samples from the chosen probability distribution. This section will introduce the basics of this process and demonstrate it with some straightforward examples. The next two sections will introduce more complex approaches to sampling before Section 13.6 develops the approach for the general multidimensional case. In Chapters 14, 15, and 16, we'll then see how to use these techniques to generate samples from the distributions defined by BSDFs, light sources, cameras, and scattering media.

13.3.1 THE INVERSION METHOD

The inversion method uses one or more uniform random variables and maps them to random variables from the desired distribution. To explain how this process works in general, we will start with a simple discrete example. Suppose we have a process with four possible outcomes. The probabilities of each of the four outcomes are given by p_1 , p_2 , p_3 , and p_4 , respectively, with the requirement that $\sum_{i=1}^4 p_i = 1$. The corresponding PDF is shown in Figure 13.1.

In order to draw a sample from this distribution, we first find the CDF $P(x)$. In the continuous case, P is the indefinite integral of p . In the discrete case, we can directly construct the CDF by stacking the bars on top of each other, starting at the left. This idea

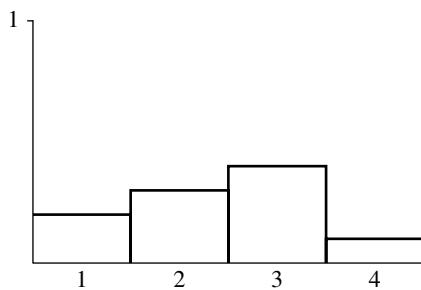


Figure 13.1: A Discrete PDF for Four Events, Each with a Probability p_i . The sum of their probabilities $\sum_i p_i$ is necessarily 1.

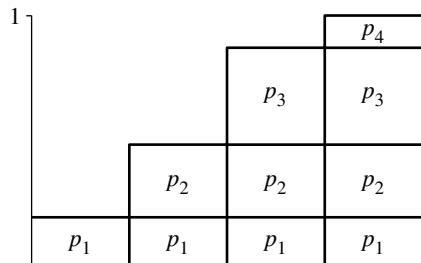
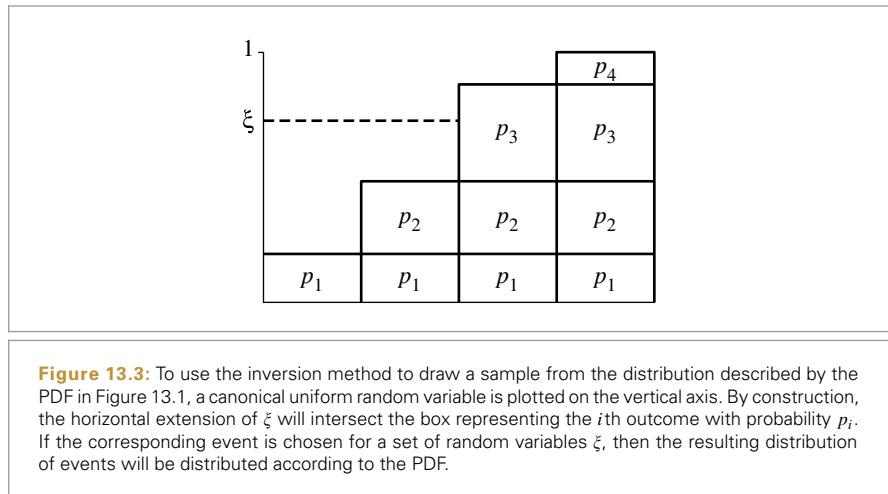


Figure 13.2: A Discrete CDF, Corresponding to the PDF in Figure 13.1. Each column's height is given by the PDF for the event that it represents plus the sum of the PDFs for the previous events, $P_i = \sum_{j=1}^i p_j$.

is shown in Figure 13.2. Notice that the height of the rightmost bar must be 1 because of the requirement that all probabilities sum to 1.

To draw a sample from the distribution, we then take a uniform random number ξ and use it to select one of the possible outcomes using the CDF, doing so in a way that chooses a particular outcome with probability equal to the outcome's own probability. This idea is illustrated in Figure 13.3, where the events' probabilities are projected onto the vertical axis and a random variable ξ selects among them. It should be clear that this draws from the correct distribution—the probability of the uniform sample hitting any particular bar is exactly equal to the height of that bar. In order to generalize this technique to continuous distributions, consider what happens as the number of discrete possibilities approaches infinity. The PDF from Figure 13.1 becomes a smooth curve, and the CDF from Figure 13.2 becomes its integral. The projection process is still the same, although if the function is continuous, the projection has a convenient mathematical



interpretation—it represents inverting the CDF and evaluating the inverse at ξ . This technique is thus called the *inversion method*.

More precisely, we can draw a sample X_i from an arbitrary PDF $p(x)$ with the following steps:

1. Compute the CDF⁵ $P(x) = \int_0^x p(x') dx'$.
2. Compute the inverse $P^{-1}(x)$.
3. Obtain a uniformly distributed random number ξ .
4. Compute $X_i = P^{-1}(\xi)$.

Example: Power Distribution

As an example of how this procedure works, consider the task of drawing samples from a *power distribution*, $p(x) \propto x^n$. The PDF of the power distribution is

$$p(x) = cx^n,$$

for the constant c that normalizes the PDF. The first task to tackle is to find the PDF. In most cases, this simply involves computing the value of the proportionality constant c , which can be found using the constraint that $\int p(x) dx = 1$:

$$\begin{aligned} \int_0^1 cx^n dx &= 1 \\ c \left. \frac{x^{n+1}}{n+1} \right|_0^1 &= 1 \\ \frac{c}{n+1} &= 1 \\ c &= n+1. \end{aligned}$$

⁵ In general, the lower limit of integration should be $-\infty$, although if $p(x) = 0$ for $x < 0$, this equation is equivalent.

Therefore, $p(x) = (n + 1)x^n$. We can integrate this PDF to get the CDF:

$$P(x) = \int_0^x p(x') dx' = x^{n+1},$$

and inversion is simple: $P^{-1}(x) = \sqrt[n+1]{x}$. Therefore, given a uniform random variable ξ , samples can be drawn from the power distribution as

$$X = \sqrt[n+1]{\xi}. \quad [13.4]$$

Another approach is to use a sampling trick that works only for the power distribution, selecting $X = \max(\xi_1, \xi_2, \dots, \xi_{n+1})$. This random variable is distributed according to the power distribution as well. To see why, note that $Pr\{X < x\}$ is the probability that *all* the $\xi_i < x$. But the ξ_i are independent, so

$$Pr\{X < x\} = \prod_{i=1}^{n+1} Pr\{\xi_i < x\} = x^{n+1},$$

which is exactly the desired CDF. Depending on the speed of your random number generator, this technique can be faster than the inversion method for small values of n .

Example: Exponential Distribution

When rendering images with participating media, it is frequently useful to draw samples from a distribution $p(x) \propto e^{-ax}$. As before, the first step is to normalize this distribution so that it integrates to one. In this case, we'll assume for now that the range of values x we'd like the generated samples to cover is $[0, \infty)$ rather than $[0, 1]$, so

$$\int_0^\infty ce^{-ax} dx = -\frac{c}{a}e^{-ax}\Big|_0^\infty = \frac{c}{a} = 1.$$

Thus we know that $c = a$, and our PDF is $p(x) = ae^{-ax}$. Now, we integrate to find $P(x)$:

$$P(x) = \int_0^x ae^{-ax'} dx' = 1 - e^{-ax}.$$

This function is easy to invert:

$$P^{-1}(x) = -\frac{\ln(1-x)}{a},$$

and we can draw samples thus:

$$X = -\frac{\ln(1-\xi)}{a}.$$

It may be tempting to simplify the log term from $\ln(1 - \xi)$ to $\ln \xi$, under the theory that because $\xi \in [0, 1)$, these are effectively the same and a subtraction can thus be saved. The problem with this idea is that ξ may have the value 0 but never has the value 1. With the simplification, it's possible that we'd try to take the logarithm of 0, which is undefined; this danger is avoided with the first formulation.⁶ While a ξ value of 0 may seem very

⁶ Once again: a subtlety that the authors didn't appreciate in the first two editions of the book.

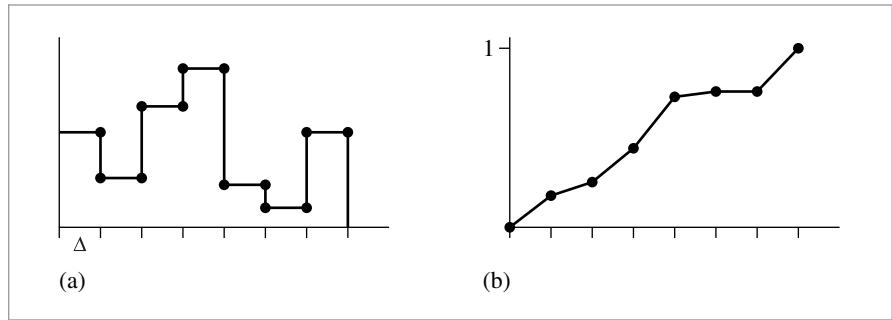


Figure 13.4: (a) Probability density function for a piecewise-constant 1D function and (b) cumulative distribution function defined by this PDF.

unlikely, it does happen (especially in the world of floating-point arithmetic, rather than the real numbers). Some of the low-discrepancy sampling patterns introduced in Chapter 7 are particularly prone to generating the value 0.

Example: Piecewise-Constant 1D Functions

An interesting exercise is to work out how to sample from 1D piecewise-constant functions (step functions). Without loss of generality, we will just consider piecewise-constant functions defined over $[0, 1]$.

Assume that the 1D function's domain is split into N equal-sized pieces of size $\Delta = 1/N$. These regions start and end at points $x_i = i\Delta$, where i ranges from 0 to N , inclusive. Within each region, the value of the function $f(x)$ is a constant (Figure 13.4(a)). The value of $f(x)$ is

$$f(x) = \begin{cases} v_0 & x_0 \leq x < x_1 \\ v_1 & x_1 \leq x < x_2 \\ \vdots & \end{cases}$$

The integral $\int f(x) dx$ is

$$c = \int_0^1 f(x) dx = \sum_{i=0}^{N-1} \Delta v_i = \sum_{i=0}^{N-1} \frac{v_i}{N}, \quad [13.5]$$

and so it is easy to construct the PDF $p(x)$ for $f(x)$ as $f(x)/c$. By direct application of the relevant formulae, the CDF $P(x)$ is a piecewise linear function defined at points x_i by

$$P(x_0) = 0$$

$$P(x_1) = \int_{x_0}^{x_1} p(x) dx = \frac{v_0}{Nc} = P(x_0) + \frac{v_0}{Nc}$$

$$P(x_2) = \int_{x_0}^{x_2} p(x) dx = \int_{x_0}^{x_1} p(x) dx + \int_{x_1}^{x_2} p(x) dx = P(x_1) + \frac{v_1}{Nc}$$

$$P(x_i) = P(x_{i-1}) + \frac{v_{i-1}}{Nc}.$$

Between two points x_i and x_{i+1} , the CDF is linearly increasing with slope v_i/c .

Recall that in order to sample $f(x)$ we need to invert the CDF to find the value x such that

$$\xi = \int_0^x p(x') dx' = P(x).$$

Because the CDF is monotonically increasing, the value of x must be between the x_i and x_{i+1} such that $P(x_i) \leq \xi$ and $\xi \leq P(x_{i+1})$. Given an array of CDF values, this pair of $P(x_i)$ values can be efficiently found with a binary search.

`Distribution1D` is a small utility class that represents a piecewise-constant 1D function's PDF and CDF and provides methods to perform this sampling efficiently.

```
(Sampling Declarations) ≡
struct Distribution1D {
    (Distribution1D Public Methods 758)
    (Distribution1D Public Data 758)
};
```

The `Distribution1D` constructor takes n values of a piecewise-constant function f . It makes its own copy of the function values, computes the function's CDF, and also stores the integral of the function, `funcInt`. Note that the constructor allocates $n+1$ `Floats` for the `cdf` array because if $f(x)$ has N step values, then we need to store the value of the CDF at each of the $N + 1$ values of x_i . Storing the CDF value of 1 at the end of the array is redundant but simplifies the sampling code later.

```
(Distribution1D Public Methods) ≡ 758
Distribution1D(const Float *f, int n)
: func(f, f + n), cdf(n + 1) {
    (Compute integral of step function at xi 758)
    (Transform step function integral into CDF 759)
}
```

```
(Distribution1D Public Data) ≡ 758
std::vector<Float> func, cdf;
Float funcInt;
```

```
(Distribution1D Public Methods) += 758
int Count() const { return func.size(); }
```

This constructor computes the integral of $f(x)$ using Equation (13.5). It stores the result in the `cdf` array for now so that it doesn't need to allocate additional temporary space for it.

```
(Compute integral of step function at xi) ≡ 758
cdf[0] = 0;
for (int i = 1; i < n + 1; ++i)
    cdf[i] = cdf[i - 1] + func[i - 1] / n;
```

Now that the value of the integral over all of $[0, 1]$ is stored in `cdf[n]`, this value can be copied into `funcInt` and the CDF can be normalized by dividing through all entries by this value.

```

Distribution1D 758
Distribution1D::cdf 758
Distribution1D::func 758
Float 1062
```

```
<Transform step function integral into CDF> ≡ 758
    funcInt = cdf[n];
    if (funcInt == 0) {
        for (int i = 1; i < n + 1; ++i)
            cdf[i] = Float(i) / Float(n);
    } else {
        for (int i = 1; i < n + 1; ++i)
            cdf[i] /= funcInt;
    }
```

The `Distribution1D::SampleContinuous()` method uses the given random sample u to sample from its distribution. It returns the corresponding value $x \in [0, 1]$ and the value of the PDF $p(x)$. If the optional `off` parameter is not `nullptr`, it returns the offset into the array of function values of the largest index where the CDF was less than or equal to u . (In other words, $\text{cdf}[*\text{off}] \leq u < \text{cdf}[*\text{off}+1]$.)

```
<Distribution1D Public Methods> +≡ 758
    Float SampleContinuous(Float u, Float *pdf, int *off = nullptr) const {
        <Find surrounding CDF segments and offset 759>
        if (off) *off = offset;
        <Compute offset along CDF segment 759>
        <Compute PDF for sampled offset 759>
        <Return x ∈ [0, 1] corresponding to sample 760>
    }
```

Mapping u to an interval matching the above criterion is carried out using the efficient binary search implemented in `FindInterval()` (see Appendix A for details).

```
<Find surrounding CDF segments and offset> ≡ 759, 760
    int offset = FindInterval(cdf.size(),
        [&](int index) { return cdf[index] <= u; });
```

Given the pair of CDF values that straddle u , we can compute x . First, we determine how far u is between $\text{cdf}[\text{offset}]$ and $\text{cdf}[\text{offset}+1]$, du , where du is 0 if $u == \text{cdf}[\text{offset}]$ and goes up to 1 if $u == \text{cdf}[\text{offset}+1]$. Because the CDF is piecewise linear, the sample value x is the same offset between x_i and x_{i+1} (Figure 13.4(b)).

```
<Compute offset along CDF segment> ≡ 759
    Float du = u - cdf[offset];
    if ((cdf[offset + 1] - cdf[offset]) > 0)
        du /= (cdf[offset + 1] - cdf[offset]);
```

The PDF for this sample $p(x)$ is easily computed since we have the function's integral in `funcInt`. (Note that the offset `offset` into the CDF array has been computed in a way so that `func[offset]` gives the value of the function in the CDF range that the sample landed in.)

```
<Compute PDF for sampled offset> ≡ 759
    if (pdf) *pdf = func[offset] / funcInt;
```

Finally, the appropriate value of x is computed and returned.

```
(Return  $x \in [0, 1]$  corresponding to sample)  $\equiv$  759
    return (offset + du) / Count();
```

In a small overloading of semantics, `Distribution1D` can also be used for discrete 1D probability distributions where there are some number of buckets n , each with some weight, and we'd like to sample among the buckets with probability proportional to their relative weights. This functionality is used, for example, by some of the Integrators in that compute a discrete distribution for the light sources in the scene with weights given by the lights' powers. Sampling from the discrete distribution just requires figuring out which pair of CDF values the sample value lies between; the PDF is computed as the discrete probability of sampling the corresponding bucket.

```
(Distribution1D Public Methods)  $\equiv$  758
int SampleDiscrete(Float u, Float *pdf = nullptr,
    Float *uRemapped = nullptr) const {
    Find surrounding CDF segments and offset 759
    if (pdf) *pdf = func[offset] / (funcInt * Count());
    if (uRemapped)
        *uRemapped = (u - cdf[offset]) / (cdf[offset + 1] - cdf[offset]);
    return offset;
}
```

It's also useful to be able to compute the PDF for sampling a given value from the discrete PDF.

```
(Distribution1D Public Methods)  $\equiv$  758
Float DiscretePDF(int index) const {
    return func[index] / (funcInt * Count());
```

13.3.2 THE REJECTION METHOD

For some functions $f(x)$, it may not be possible to integrate them in order to find their PDFs, or it may not be possible to analytically invert their CDFs. The *rejection method* is a technique for generating samples according to a function's distribution without needing to do either of these steps; it is essentially a dart-throwing approach. Assume that we want to draw samples from some such function $f(x)$ but we do have a PDF $p(x)$ that satisfies $f(x) < c p(x)$ for some scalar constant c , and suppose that we do know how to sample from p . The rejection method is then:

```
loop forever:
    sample  $X$  from  $p$ 's distribution
    if  $\xi < f(X)/(c p(X))$  then
        return  $X$ 
```

This procedure repeatedly chooses a pair of random variables (X, ξ) . If the point $(X, \xi | c p(X))$ lies under $f(X)$, then the sample X is accepted. Otherwise, it is rejected

```
Distribution1D 758
Distribution1D::Count() 758
Distribution1D::func 758
Distribution1D::funcInt 758
Float 1062
```

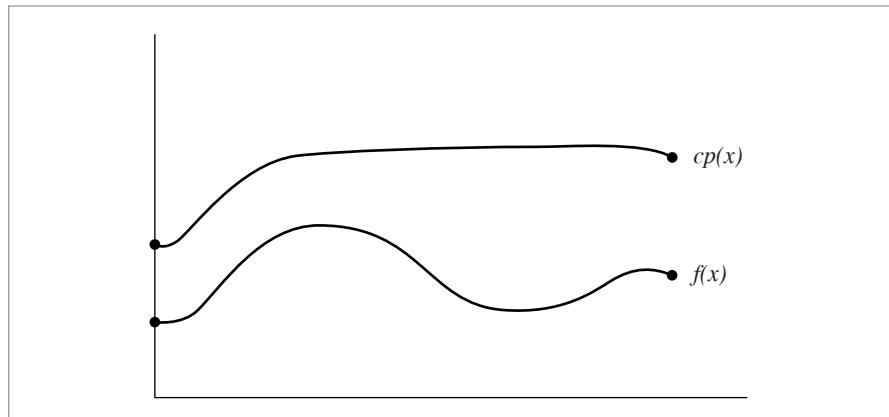


Figure 13.5: Rejection sampling generates samples according to the distribution of an arbitrary function $f(x)$ even if f 's PDF is unknown or its CDF can't be inverted. If some distribution $p(x)$ and a scalar constant c are known such that $f(x) < c p(x)$, then samples can be drawn from $p(x)$ and randomly accepted with the rejection method. The closer the fit of $c p(x)$ to $f(x)$, the more efficient this process is.

and a new sample pair is chosen. This idea is illustrated in Figure 13.5. Without going into too much detail, it should be clear that the efficiency of this scheme depends on how tightly $c p(x)$ bounds $f(x)$. This technique works in any number of dimensions.

Rejection sampling isn't actually used in any of the Monte Carlo algorithms currently implemented in pbrt. We will normally prefer to find distributions that are similar to $f(x)$ that can be sampled directly, so that well-distributed points on $[0, 1]^n$ can be mapped to sample points that are in turn well-distributed, as will be discussed in Section 13.8. Nevertheless, rejection sampling is an important technique to be aware of, particularly when debugging Monte Carlo implementations. For example, if one suspects the presence of a bug in code that draws samples from some distribution using the inversion method, then one can replace it with a straightforward implementation based on the rejection method and see if the Monte Carlo estimator computes the same result. Of course, it's necessary to take many samples in situations like these, so that variance in the estimates doesn't mask errors.

Example: Rejection Sampling a Unit Circle

Suppose we want to select a uniformly distributed point inside a unit circle. Using the rejection method, we simply select a random (x, y) position inside the circumscribed square and return it if it falls inside the circle. This process is shown in Figure 13.6.

The function `RejectionSampleDisk()` implements this algorithm. A similar approach will work to generate uniformly distributed samples on the inside of any complex shape as long as it has an inside–outside test.

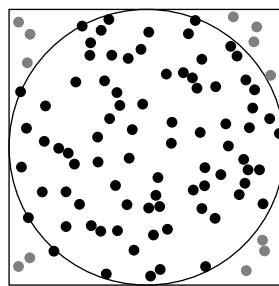


Figure 13.6: Rejection Sampling a Circle. One approach to finding uniform points in the unit circle is to sample uniform random points in the unit square and reject all that lie outside the circle. The remaining points will be uniformly distributed within the circle.

(Sampling Function Definitions) +≡

```
Point2f RejectionSampleDisk(RNG &rng) {
    Point2f p;
    do {
        p.x = 1 - 2 * rng.UniformFloat();
        p.y = 1 - 2 * rng.UniformFloat();
    } while (p.x * p.x + p.y * p.y > 1);
    return p;
}
```

In general, the efficiency of rejection sampling depends on the percentage of samples that are expected to be rejected. For the problem of finding uniform points in the 2D case, this is easy to compute. It is the area of the circle divided by the area of the square: $\frac{\pi}{4} \approx 78.5\%$. If the method is applied to generate samples in hyperspheres in the general n -dimensional case, however, the volume of an n -dimensional hypersphere actually goes to 0 as n increases, and this approach becomes increasingly inefficient.

* 13.4 METROPOLIS SAMPLING

Metropolis sampling is a technique for generating a set of samples from a non-negative function f that is distributed proportionally to f 's value (Metropolis et al. 1953).⁷ Remarkably, it is able to do so without requiring anything more than the ability to evaluate f ; it is not necessary to be able to integrate f , normalize the integral, and invert the resulting CDF. Furthermore, *every* iteration produces a usable sample generated from the function's PDF; Metropolis sampling doesn't share the shortcoming of rejection sam-

Point2f 68

RNG 1065

RNG::UniformFloat() 1066

⁷ We will refer to the Monte Carlo sampling algorithm as “the Metropolis algorithm” here. Other commonly used shorthands for it include M(RT)², for the initials of the authors of the original paper, and Metropolis-Hastings, which gives a nod to Hastings, who generalized the technique (Fishman 1996). The term *Markov chain Monte Carlo* is synonymous with Metropolis sampling and derived techniques.

pling that the number of iterations needed to obtain the next sample cannot be bounded. It can thus efficiently generate samples from a wider variety of functions than the techniques introduced in the previous section. It forms the foundation of the Metropolis light transport algorithm implemented in Section 16.4.

Metropolis sampling does have a few disadvantages: successive samples in the sequence are statistically correlated, and it is thus not possible to ensure that a small set of samples generated by Metropolis is well distributed across the domain. It's only in the limit over a large number of samples that the samples will cover the domain. As such, the variance reduction advantages of techniques like stratified sampling (Section 13.8.1) are generally not available when using Metropolis sampling.

13.4.1 BASIC ALGORITHM

More concretely, the Metropolis algorithm generates a set of samples X_i from a function f , which is defined over an arbitrary-dimensional state space Ω (frequently, $\Omega = \mathbb{R}^n$) and returns a value in the reals, $f: \Omega \rightarrow \mathbb{R}$. After the first sample $X_0 \in \Omega$ has been selected, each subsequent sample X_i is generated by using a random *mutation* to X_{i-1} to compute a proposed sample X' . The mutation may be accepted or rejected, and X_i is accordingly set to either X' or X_{i-1} . When these transitions from one state to another are chosen subject to a few requirements (to be described shortly), the distribution of X_i values that results reaches an equilibrium distribution; this distribution is the *stationary distribution*. In the limit, the distribution of the set of samples $X_i \in \Omega$ is proportional to $f(x)$'s probability density function $p(x) = f(x)/\int_{\Omega} f(x)d\Omega$.

In order to generate the correct distribution of samples, it is necessary to generate proposed mutations and then accept or reject the mutations subject to a few constraints. Assume that we have a mutation method that proposes changing from a given state X into a new state X' (this might be done by perturbing X in some way, or even by generating a completely new value). We must be able to compute a tentative transition function $T(X \rightarrow X')$ that gives the probability density of the mutation technique's proposing a transition to X' , given that the current state is X . (Section 13.4.2 will discuss considerations for designing transition functions.)

Given a transition function, it is possible to define an *acceptance probability* $a(X \rightarrow X')$ that gives the probability of accepting a proposed mutation from X to X' in a way that ensures that the distribution of samples is proportional to $f(x)$. If the distribution is already in equilibrium, the transition density between any two states must be equal:⁸

$$f(X) T(X \rightarrow X') a(X \rightarrow X') = f(X') T(X' \rightarrow X) a(X' \rightarrow X). \quad (13.6)$$

This property is called *detailed balance*.

Since f and T are set, Equation (13.6) tells us how a must be defined. In particular, a definition of a that maximizes the rate at which equilibrium is reached is

$$a(X \rightarrow X') = \min \left(1, \frac{f(X') T(X' \rightarrow X)}{f(X) T(X \rightarrow X')} \right). \quad (13.7)$$

⁸ See Kalos and Whitlock (1986) or Veach's thesis (1997) for a rigorous derivation.

One thing to immediately notice from Equation (13.7) is that, if the transition probability density is the same in both directions, the acceptance probability simplifies to

$$a(X \rightarrow X') = \min \left(1, \frac{f(X')}{f(X)} \right). \quad [13.8]$$

Put together, we have the basic Metropolis sampling algorithm in pseudocode:

```
X = X0
for i = 1 to n
    X' = mutate(X)
    a = accept(X, X')
    if (random() < a)
        X = X'
    record(X)
```

This code generates n samples by mutating the previous sample and computing acceptance probabilities as in Equation (13.7). Each sample X_i can then be recorded in a data structure or used as a sample for integration.

Because the Metropolis algorithm naturally avoids parts of Ω where $f(x)$'s value is relatively low, few samples will be accumulated there. In order to get some information about $f(x)$'s behavior in such regions, the *expected values* technique can be used to enhance the basic Metropolis algorithm. In this case, we still decide which state to transition into as before, but we record a sample at each of X and X' , regardless of which one is selected by the acceptance criteria. Each of these recorded samples has a weight associated with it, where the weights are the probabilities $(1 - a)$ for X and a for X' , where a is the acceptance probability. Expected values doesn't change the way we decide which state, X or X' , to use at the next step; that part of the computation remains the same.

Updated pseudocode shows the idea:

```
X = X0
for i = 1 to n
    X' = mutate(X)
    a = accept(X, X')
    record(X, 1 - a)
    record(X', a)
    if (random() < a)
        X = X'
```

Comparing the two pieces of pseudocode, we can see that in the limit, the same weight distribution will be accumulated for X and X' . Expected values more quickly give a smoother result and more information about the areas where $f(x)$ is low than the basic algorithm does.

13.4.2 CHOOSING MUTATION STRATEGIES

In general, one has a lot of freedom in choosing mutation strategies, subject to being able to compute the tentative transition density $T(X \rightarrow X')$. Recall from Equation (13.8) that if the transition densities are symmetric then it is not even necessary to be able to

compute them to apply the Metropolis sampling algorithm. It's easy to apply multiple mutation strategies, so if there are some that are effective in only some circumstances it doesn't hurt to try using them as one of a set of approaches.

It is generally desirable that mutations propose large changes to the current sample rather than small ones. Doing so more quickly explores the state space rather than letting the sampler get stuck in a small region of it. However, when the function's value $f(X)$ is relatively large at the current sample X , then it is likely that many proposed mutations will be rejected (consider the case where $f(X) \gg f(X')$ in Equation (13.8); $a(X \rightarrow X')$ will be very small). We'd like to avoid the case where many samples in a row are the same, again to better explore new parts of the state space: staying in the same state for many samples in a row leads to increased variance—intuitively, it makes sense that the more we move around Ω , the better the overall results will be. For this case, small mutations are likely to propose samples X' where f is still relatively large, leading to higher acceptance properties.

Thus, one useful mutation approach is to apply random perturbations to the current sample X . If the sample X is represented by a vector of real numbers (x_0, x_1, \dots) , then some or all of the sample dimensions x_i can be perturbed. One possibility is to perturb them by adding or subtracting a scaled random variable:

$$x'_i = (x_i \pm s \xi) \bmod 1$$

for some scale factor s and where the mod operator wraps values around the boundaries to remain in $[0, 1]$. This method is symmetric, so we don't need to compute the transition densities $T(X \rightarrow X')$ when using it with Metropolis sampling.

A related mutation approach is to just discard the current sample entirely and generate a new one with uniform random numbers:

$$x_i = \xi.$$

(Note that this is also a symmetric method.) Occasionally generating a completely new sample in this manner is important since it ensures that we don't get stuck in one part of the state space and never sample the rest of it. In general, it's necessary that it be possible to reach all states $X \in \Omega$ where $f(X) > 0$ with nonzero probability (this property is called *ergodicity*). In particular, to ensure ergodicity it suffices that $T(X \rightarrow X') > 0$ for all X and X' where $f(X) > 0$ and $f(X') > 0$.

Another approach is to use PDFs that match some part of the function being sampled. If we have a PDF $p(x)$ that is similar to some component of f , then we can use that to derive a mutation strategy by just drawing new samples $X \sim p$. In this case, the transition function is straightforward:

$$T(X \rightarrow X') = p(X').$$

In other words, the current state X doesn't matter for computing the transition density: we propose a transition into a state X' with a density that depends only on the newly proposed state X' and not at all on the current state.

13.4.3 START-UP BIAS

One issue that we've sidestepped thus far is how the initial sample X_0 is computed. The transition and acceptance methods above tell us how to generate new samples X_{i+1} , but all presuppose that the current sample X_i has itself *already* been sampled with probability proportional to f . Using a sample not from f 's distribution leads to a problem called *start-up bias*.

A common solution to this problem is to run the Metropolis sampling algorithm for some number of iterations from an arbitrary starting state, discard the samples that are generated, and then start the process for real, assuming that that has brought us to an appropriately sampled X value. This is unsatisfying for two reasons: first, the expense of taking the samples that were then discarded may be high, and, second, we can only guess at how many initial samples must be taken in order to remove start-up bias.

An alternative approach can be used if another sampling method is available: an initial value X_0 is sampled using any density function $X_0 \sim p(x)$. We start the Markov chain from the state X_0 , but we weight the contributions of all of the samples that we generate by the weight

$$w = \frac{f(X_0)}{p(X_0)}.$$

This method eliminates start-up bias completely and does so in a predictable manner.

The only potential problem comes if $f(X_0) = 0$ for the X_0 we chose; in this case, all samples will have a weight of zero! This doesn't mean that the algorithm is biased, however; the expected value of the result still converges to the correct distribution (see Veach (1997) for further discussion and for a proof of the correctness). To reduce variance and avoid this risk, we can instead sample a set of N candidate sample values, Y_1, \dots, Y_N , defining a weight for each by

$$w_i = \frac{f(Y_i)}{p(Y_i)}.$$

We then choose the starting X_0 sample for the Metropolis algorithm from the Y_i with probability proportional to their relative weights and compute a sample weight w as the average of all of the w_i weights. All subsequent samples X_i that are generated by the Metropolis algorithm are then weighted by the sample weight w .

13.4.4 1D SETTING

In order to illustrate some of the ideas in this section, we'll show how Metropolis sampling can be used to sample a simple 1D function, defined over $\Omega = [0, 1]$ and 0 everywhere else (see Figure 13.7).

$$f(x) = \begin{cases} (x - 0.5)^2 & 0 \leq x \leq 1 \\ 0 & \text{otherwise.} \end{cases} \quad [13.9]$$

For this example, assume that we don't actually know the exact form of f —it's just a black box that we can evaluate at particular x values. (Clearly, if we knew that f was just

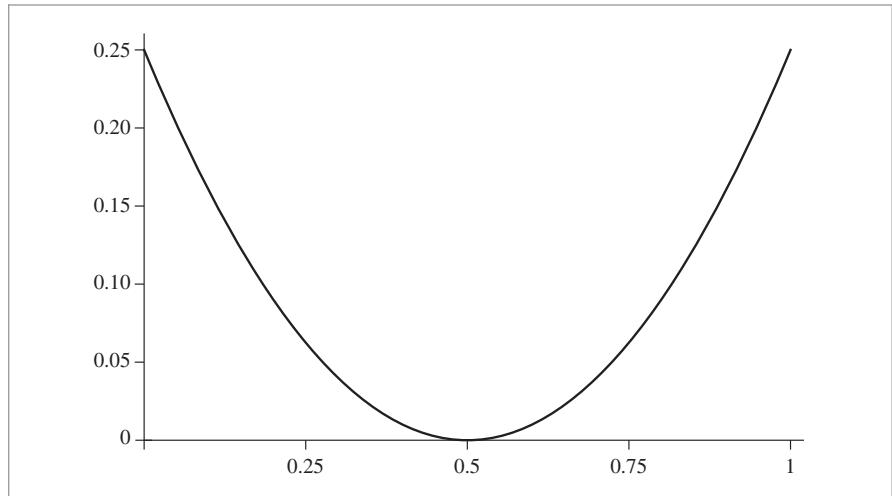


Figure 13.7: Graph of the function $f(x)$ used to illustrate Metropolis sampling in this section.

Equation (13.9), there'd be no need for Metropolis in order to draw samples from its distribution!)

We'll define two mutation strategies based on the ideas introduced in Section 13.4.2 and randomly choose among them each time a mutation is proposed, according to a desired distribution of how frequently each is to be used.

The first strategy, mutate_1 , discards the current sample X and uniformly samples a new one, X' , from the entire state space $[0, 1]$. The transition function for this mutation is straightforward. For mutate_1 , since we are uniformly sampling over $[0, 1]$, the probability density is uniform over the entire domain; in this case, the density is just one everywhere. We have

$$\text{mutate}_1(X) \rightarrow \xi$$

$$T_1(X \rightarrow X') = 1.$$

The second mutation adds a random offset between ± 0.05 to the current sample X in an effort to sample repeatedly in the parts of f that make a high contribution to the overall distribution. The transition probability density is 0 if X and X' are far enough away that mutate_2 will never mutate from one to the other; otherwise, the density is constant. Normalizing the density so that it integrates to 1 over its domain gives the value $1/0.1$. Both this and mutate_1 are symmetric, so the transition densities aren't needed to implement the sampling algorithm.

$$\text{mutate}_2(X) \rightarrow X + 0.1(\xi - 0.5)$$

$$T_2(X \rightarrow X') = \begin{cases} \frac{1}{0.1} & |X - X'| \leq 0.05 \\ 0 & \text{otherwise.} \end{cases}$$

To find the initial sample, we only need to take a single sample with a uniform PDF over Ω , since $f(x) > 0$ except for a single point in Ω for which there is zero probability of sampling:

$$X_0 = \xi.$$

The sample weight w is then just $f(X_0)$.

We can now run the Metropolis algorithm and generate samples X_i of f . At each transition, we have two weighted samples to record (recall the expected values pseudocode from Section 13.4.1). A simple approach for reconstructing the approximation to f 's probability distribution is to store sums of the weights in a set of buckets of uniform width; each sample falls in a single bucket and contributes to it. Figure 13.8 shows some results. For both graphs, a chain of 10,000 mutations was followed, with the sample weights accumulated in 50 buckets over $[0, 1]$.

In the top graph, only `mutate1` was used. This alone isn't a very effective mutation, since it doesn't take advantage of the times when it has found a sample in a region of Ω where f has a relatively large value to generate additional samples in that neighborhood. However, the graph does suggest that the algorithm is converging to the correct distribution.

On the bottom, one of `mutate1` and `mutate2` was randomly chosen, with probabilities of 10% and 90%, respectively. We see that for the same number of samples taken, we converge to f 's distribution with less variance. This is because the algorithm is more effectively able to concentrate its work in areas where f 's value is large, proposing fewer mutations to parts of state space where f 's value is low. For example, if $X = .8$ and the second mutation proposes $X' = .75$, this will be accepted $f(.75)/f(.8) \approx 69\%$ of the time, while mutations from $.75$ to $.8$ will be accepted $\min(1, 1.44) = 100\%$ of the time. Thus, we see how the algorithm naturally tends to try to avoid spending time sampling around the dip in the middle of the curve.

One important thing to note about these graphs is that the y axis has units that are different from those in Figure 13.7, where f is graphed. Recall that Metropolis sampling provides a set of samples distributed according to f 's probability density; as such (for example), we would get the same sample distribution for another function $g = 2f$. If we wish to reconstruct an approximation to f directly from Metropolis samples, we must compute a normalization factor and use it to scale the PDF.

Figure 13.9 shows the surprising result of only using `mutate2` to propose sample values. On the left, 10,000 samples were taken using just that mutation. Clearly, things have gone awry—no samples $X_i > .5$ were generated and the result doesn't bear much resemblance to f .

Thinking about the acceptance probability again, we can see that it would take a large number of mutations, each with low probability of acceptance, to move X_i down close enough to $.5$ such that `mutate2`'s short span would be enough to move over to the other side. Since the Metropolis algorithm tends to stay away from the lower valued regions of f (recall the comparison of probabilities for moving from $.8$ to $.75$ versus moving from $.75$ to $.8$), this happens quite rarely. The right side of Figure 13.9 shows what happens

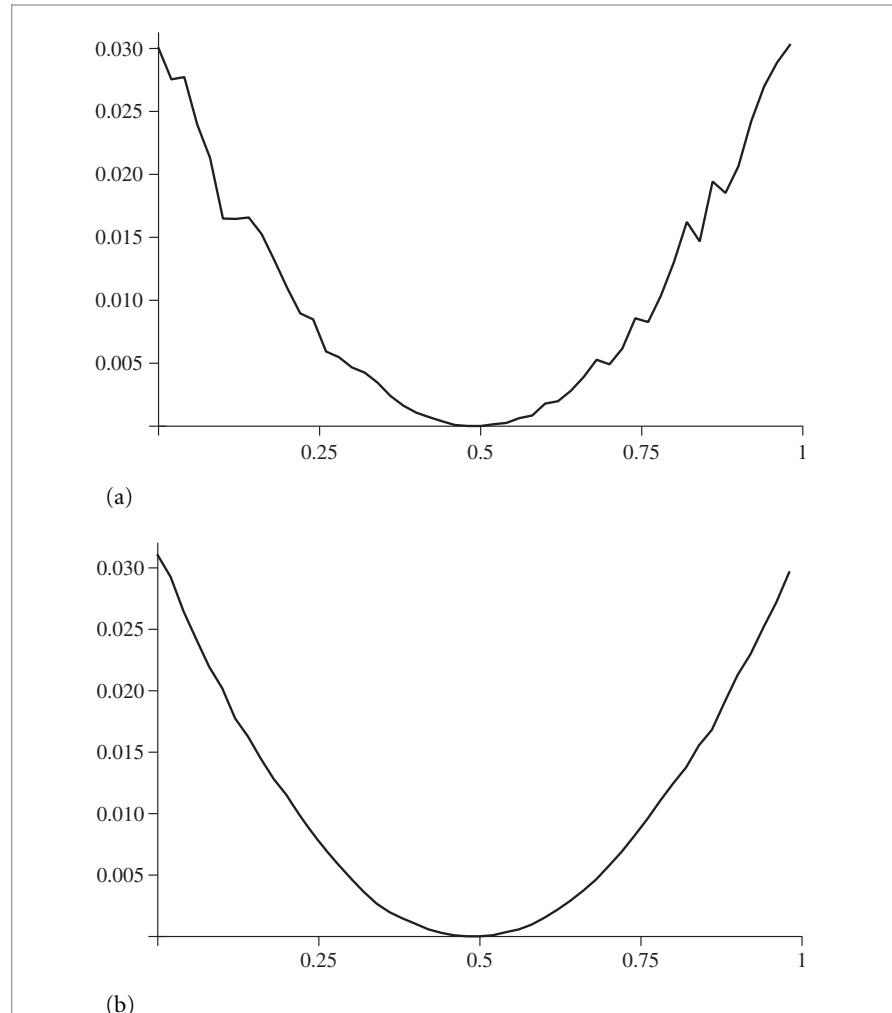


Figure 13.8: Comparison of Metropolis Sampling Strategies. (a) Only `mutate1` was used, randomly selecting a completely new value at each step. (b) Both `mutate1` and `mutate2` were used, with a 1:9 ratio. For the same number of samples, variance is substantially lower, thanks to a higher rate of acceptance of `mutate2`'s proposed mutations.

if 300,000 samples are taken. This was enough to be able to jump from one side of .5 to the other a few times, but not enough to get close to the correct distribution. Using `mutate2` alone is thus not mathematically incorrect, just inefficient: it does have nonzero probability of proposing a transition from any state to any other state (through a chain of multiple transitions).

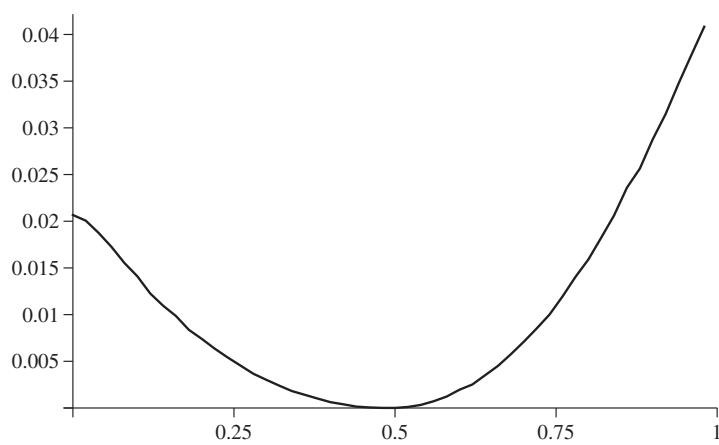
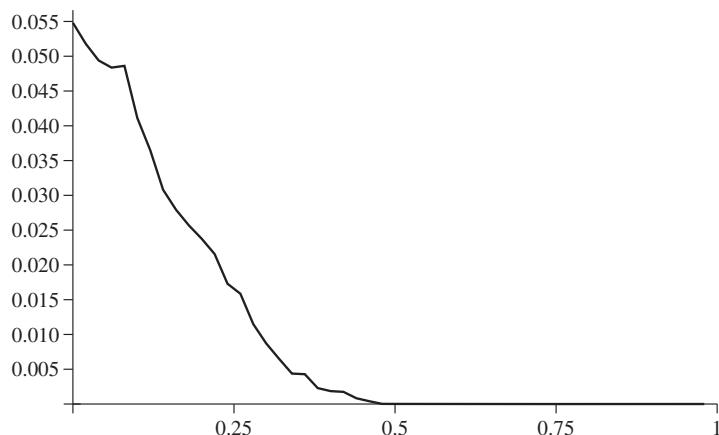


Figure 13.9: Why It Is Important to Periodically Pick a Completely New Sample Value with Metropolis Sampling. (a) 10,000 iterations using only `mutate2` were computed, (b) 300,000 iterations. It is very unlikely that a series of mutations will be able to move from one side of the curve, across .5, to the other side, since mutations that move toward areas where f 's value is low will usually be rejected. As such, the results are inaccurate for these numbers of iterations. (It's small solace that they would be correct in the limit.)

13.4.5 ESTIMATING INTEGRALS WITH METROPOLIS SAMPLING

We can apply the Metropolis algorithm to estimating integrals such as $\int f(x)g(x) d\Omega$. Doing so is the basis of the Metropolis light transport algorithm implemented in Section 16.4.

To see how the samples from Metropolis sampling can be used in this way, recall that the standard Monte Carlo estimator, Equation (13.3), says that

$$\int_{\Omega} f(x)g(x) d\Omega \approx \frac{1}{N} \sum_{i=1}^N \frac{f(X_i)g(X_i)}{p(X_i)},$$

where X_i are sampled from a density function $p(x)$. Thus, if we apply Metropolis sampling and generate a set of samples, X_1, \dots, X_N , from a density function that is proportional to $f(x)$, then we can estimate this integral as

$$\int_{\Omega} f(x)g(x) d\Omega \approx \left[\frac{1}{N} \sum_{i=1}^N g(X_i) \right] \cdot \int_{\Omega} f(x) d\Omega. \quad (13.10)$$

13.5 TRANSFORMING BETWEEN DISTRIBUTIONS

In describing the inversion method, we introduced a technique that generates samples according to some distribution by transforming canonical uniform random variables in a particular manner. Here, we will investigate the more general question of which distribution results when we transform samples from an arbitrary distribution to some other distribution with a function f .

Suppose we are given random variables X_i that are already drawn from some PDF $p_x(x)$. Now, if we compute $Y_i = y(X_i)$, we would like to find the distribution of the new random variable Y_i . This may seem like an esoteric problem, but we will see that understanding this kind of transformation is critical for drawing samples from multidimensional distribution functions.

The function $y(x)$ must be a one-to-one transformation; if multiple values of x mapped to the same y value, then it would be impossible to unambiguously describe the probability density of a particular y value. A direct consequence of y being one-to-one is that its derivative must either be strictly greater than 0 or strictly less than 0, which implies that

$$Pr\{Y \leq y(x)\} = Pr\{X \leq x\},$$

and therefore

$$P_y(y) = P_y(y(x)) = P_x(x).$$

This relationship between CDFs leads directly to the relationship between their PDFs. If we assume that y 's derivative is greater than 0, differentiating gives

$$p_y(y) \frac{dy}{dx} = p_x(x),$$

and so

$$p_y(y) = \left(\frac{dy}{dx} \right)^{-1} p_x(x).$$

In general, y 's derivative is either strictly positive or strictly negative, and the relationship between the densities is

$$p_y(y) = \left| \frac{dy}{dx} \right|^{-1} p_x(x).$$

How can we use this formula? Suppose that $p_x(x) = 2x$ over the domain $[0, 1]$, and let $Y = \sin X$. What is the PDF of the random variable Y ? Because we know that $dy/dx = \cos x$,

$$p_y(y) = \frac{p_x(x)}{|\cos x|} = \frac{2x}{\cos x} = \frac{2 \arcsin y}{\sqrt{1 - y^2}}.$$

This procedure may seem backward—usually we have some PDF that we want to sample from, not a given transformation. For example, we might have X drawn from some $p_x(x)$ and would like to compute Y from some distribution $p_y(y)$. What transformation should we use? All we need is for the CDFs to be equal, or $P_y(y) = P_x(x)$, which immediately gives the transformation

$$y(x) = P_y^{-1}(P_x(x)).$$

This is a generalization of the inversion method, since if X were uniformly distributed over $[0, 1]$ then $P_x(x) = x$, and we have the same procedure as was introduced previously.

13.5.1 TRANSFORMATION IN MULTIPLE DIMENSIONS

In the general n -dimensional case, a similar derivation gives the analogous relationship between different densities. We will not show the derivation here; it follows the same form as the 1D case. Suppose we have an n -dimensional random variable X with density function $p_x(x)$. Now let $Y = T(X)$, where T is a bijection. In this case, the densities are related by

$$p_y(y) = p_y(T(x)) = \frac{p_x(x)}{|J_T(x)|},$$

where $|J_T|$ is the absolute value of the determinant of T 's Jacobian matrix, which is

$$\begin{pmatrix} \partial T_1 / \partial x_1 & \cdots & \partial T_1 / \partial x_n \\ \vdots & \ddots & \vdots \\ \partial T_n / \partial x_1 & \cdots & \partial T_n / \partial x_n \end{pmatrix},$$

where T_i are defined by $T(x) = (T_1(x), \dots, T_n(x))$.

13.5.2 POLAR COORDINATES

The polar transformation is given by

$$\begin{aligned} x &= r \cos \theta \\ y &= r \sin \theta. \end{aligned}$$

Suppose we draw samples from some density $p(r, \theta)$. What is the corresponding density $p(x, y)$? The Jacobian of this transformation is

$$J_T = \begin{pmatrix} \frac{\partial x}{\partial r} & \frac{\partial x}{\partial \theta} \\ \frac{\partial y}{\partial r} & \frac{\partial y}{\partial \theta} \end{pmatrix} = \begin{pmatrix} \cos \theta & -r \sin \theta \\ \sin \theta & r \cos \theta \end{pmatrix},$$

and the determinant is $r (\cos^2 \theta + \sin^2 \theta) = r$. So $p(x, y) = p(r, \theta)/r$. Of course, this is backward from what we usually want—typically we start with a sampling strategy in Cartesian coordinates and want to transform it to one in polar coordinates. In that case, we would have

$$p(r, \theta) = r p(x, y).$$

13.5.3 SPHERICAL COORDINATES

Given the spherical coordinate representation of directions,

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

the Jacobian of this transformation has determinant $|J_T| = r^2 \sin \theta$, so the corresponding density function is

$$p(r, \theta, \phi) = r^2 \sin \theta p(x, y, z).$$

This transformation is important since it helps us represent directions as points (x, y, z) on the unit sphere. Remember that solid angle is defined as the area of a set of points on the unit sphere. In spherical coordinates, we previously derived

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

So if we have a density function defined over a solid angle Ω , this means that

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

The density with respect to θ and ϕ can therefore be derived:

$$\begin{aligned} p(\theta, \phi) \, d\theta \, d\phi &= p(\omega) \, d\omega \\ p(\theta, \phi) &= \sin \theta \, p(\omega). \end{aligned}$$

13.6 2D SAMPLING WITH MULTIDIMENSIONAL TRANSFORMATIONS

Suppose we have a 2D joint density function $p(x, y)$ that we wish to draw samples (X, Y) from. Sometimes multidimensional densities are separable and can be expressed as the product of 1D densities—for example,

$$p(x, y) = p_x(x)p_y(y),$$

for some p_x and p_y . In this case, random variables (X, Y) can be found by independently sampling X from p_x and Y from p_y . Many useful densities aren't separable, however, so

we will introduce the theory of how to sample from multidimensional distributions in the general case.

Given a 2D density function, the *marginal density function* $p(x)$ is obtained by “integrating out” one of the dimensions:

$$p(x) = \int p(x, y) dy. \quad (13.11)$$

This can be thought of as the density function for X alone. More precisely, it is the average density for a particular x over *all* possible y values.

The *conditional density function* $p(y|x)$ is the density function for y given that some particular x has been chosen (it is read “ p of y given x ”):

$$p(y|x) = \frac{p(x, y)}{p(x)}. \quad (13.12)$$

The basic idea for 2D sampling from joint distributions is to first compute the marginal density to isolate one particular variable and draw a sample from that density using standard 1D techniques. Once that sample is drawn, one can then compute the conditional density function given that value and draw a sample from that distribution, again using standard 1D sampling techniques.

13.6.1 UNIFORMLY SAMPLING A HEMISPHERE

As an example, consider the task of choosing a direction on the hemisphere uniformly with respect to solid angle. Remember that a uniform distribution means that the density function is a constant, so we know that $p(\omega) = c$. In conjunction with the fact that the density function must integrate to one over its domain, we have

$$\int_{\mathcal{H}^2} p(\omega) d\omega = 1 \Rightarrow c \int_{\mathcal{H}^2} d\omega = 1 \Rightarrow c = \frac{1}{2\pi}.$$

This tells us that $p(\omega) = 1/(2\pi)$, or $p(\theta, \phi) = \sin \theta / (2\pi)$ (using a result from the previous example about spherical coordinates). Note that this density function is separable. Nevertheless, we will use the marginal and conditional densities to illustrate the multi-dimensional sampling technique.

Consider sampling θ first. To do so, we need θ ’s marginal density function $p(\theta)$:

$$p(\theta) = \int_0^{2\pi} p(\theta, \phi) d\phi = \int_0^{2\pi} \frac{\sin \theta}{2\pi} d\phi = \sin \theta.$$

Now, compute the conditional density for ϕ :

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

Notice that the density function for ϕ is itself uniform; this should make intuitive sense given the symmetry of the hemisphere. Now, we use the 1D inversion technique to sample each of these PDFs in turn:

$$P(\theta) = \int_0^\theta \sin \theta' d\theta' = 1 - \cos \theta$$

$$P(\phi|\theta) = \int_0^\phi \frac{1}{2\pi} d\phi' = \frac{\phi}{2\pi}.$$

Inverting these functions is straightforward, and here we can safely replace $1 - \xi$ with ξ , giving

$$\theta = \cos^{-1} \xi_1$$

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

Converting these back to Cartesian coordinates, we get the final sampling formulae:

$$x = \sin \theta \cos \phi = \cos(2\pi \xi_2) \sqrt{1 - \xi_1^2}$$

$$y = \sin \theta \sin \phi = \sin(2\pi \xi_2) \sqrt{1 - \xi_1^2}$$

$$z = \cos \theta = \xi_1.$$

This sampling strategy is implemented in the following code. Two uniform random numbers are provided in u , and a vector on the hemisphere is returned.

```
<Sampling Function Definitions> +≡
Vector3f UniformSampleHemisphere(const Point2f &u) {
    Float z = u[0];
    Float r = std::sqrt(std::max((Float)0, (Float)1. - z * z));
    Float phi = 2 * Pi * u[1];
    return Vector3f(r * std::cos(phi), r * std::sin(phi), z);
}
```

For each sampling routine like this in `pbrt`, there is a corresponding function that returns the value of the PDF for a particular sample. For such functions, it is important to be clear which PDF is being evaluated—for example, for a direction on the hemisphere, we have already seen these densities expressed differently in terms of solid angle and in terms of (θ, ϕ) . For hemispheres (and all other directional sampling), these functions return values with respect to solid angle. For the hemisphere, the solid angle PDF is a constant $p(\omega) = 1/(2\pi)$.

```
<Sampling Function Definitions> +≡
Float UniformHemispherePdf() {
    return Inv2Pi;
}
```

Float 1062
 Inv2Pi 1063
 Pi 1063
 Point2f 68
 Vector3f 60

Sampling the full sphere uniformly over its area follows almost exactly the same derivation, which we omit here. The end result is

$$x = \cos(2\pi \xi_2) \sqrt{1 - z^2} = \cos(2\pi \xi_2) 2\sqrt{\xi_1(1 - \xi_1)}$$

$$y = \sin(2\pi \xi_2) \sqrt{1 - z^2} = \sin(2\pi \xi_2) 2\sqrt{\xi_1(1 - \xi_1)}$$

$$z = 1 - 2\xi_1.$$

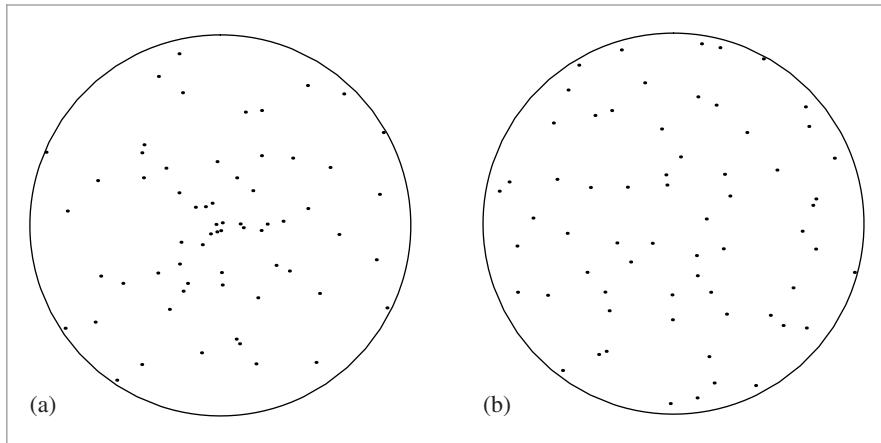


Figure 13.10: (a) When the obvious but incorrect mapping of uniform random variables to points on the disk is used, the resulting distribution is not uniform and the samples are more likely to be near the center of the disk. (b) The correct mapping gives a uniform distribution of points.

(Sampling Function Definitions) +≡

```
Vector3f UniformSampleSphere(const Point2f &u) {
    Float z = 1 - 2 * u[0];
    Float r = std::sqrt(std::max((Float)0, (Float)1 - z * z));
    Float phi = 2 * Pi * u[1];
    return Vector3f(r * std::cos(phi), r * std::sin(phi), z);
}
```

(Sampling Function Definitions) +≡

```
Float UniformSpherePdf() {
    return Inv4Pi;
}
```

13.6.2 SAMPLING A UNIT DISK

Although the disk seems a simpler shape to sample than the hemisphere, it can be trickier to sample uniformly because it has an incorrect intuitive solution. The wrong approach is the seemingly obvious one: $r = \xi_1$, $\theta = 2\pi\xi_2$. Although the resulting point is both random and inside the circle, it is *not* uniformly distributed; it actually clumps samples near the center of the circle. Figure 13.10(a) shows a plot of samples on the unit disk when this mapping was used for a set of uniform random samples (ξ_1, ξ_2) . Figure 13.10(b) shows uniformly distributed samples resulting from the following correct approach.

Since we're going to sample uniformly with respect to area, the PDF $p(x, y)$ must be a constant. By the normalization constraint, $p(x, y) = 1/\pi$. If we transform into polar coordinates (see the example in Section 13.5.2), we have $p(r, \theta) = r/\pi$. Now we compute the marginal and conditional densities as before:

Float 1062

Inv4Pi 1063

Pi 1063

Point2f 68

Vector3f 60

$$p(r) = \int_0^{2\pi} p(r, \theta) d\theta = 2r$$

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

As with the hemisphere case, the fact that $p(\theta|r)$ is a constant should make sense because of the symmetry of the circle. Integrating and inverting to find $P(r)$, $P^{-1}(r)$, $P(\theta)$, and $P^{-1}(\theta)$, we can find that the correct solution to generate uniformly distributed samples on a disk is

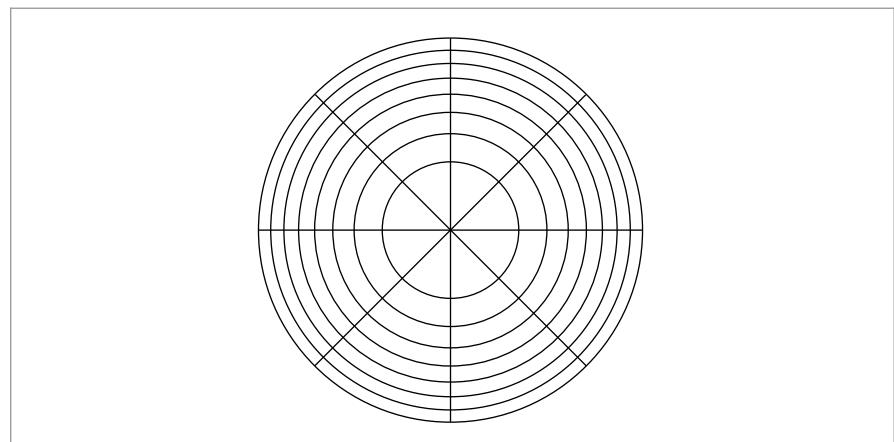
$$r = \sqrt{\xi_1}$$

$$\theta = 2\pi\xi_2.$$

Taking the square root of ξ_1 effectively pushes the samples back toward the edge of the disk, counteracting the clumping referred to earlier.

```
<Sampling Function Definitions> +≡
Point2f UniformSampleDisk(const Point2f &u) {
    Float r = std::sqrt(u[0]);
    Float theta = 2 * Pi * u[1];
    return Point2f(r * std::cos(theta), r * std::sin(theta));
}
```

Although this mapping solves the problem at hand, it distorts areas on the disk; areas on the unit square are elongated and/or compressed when mapped to the disk (Figure 13.11). (Section 13.8.3 will discuss in more detail why this distortion is a disadvantage.) A better approach is a “concentric” mapping from the unit square to the unit circle that avoids this problem. The concentric mapping takes points in the square $[-1, 1]^2$



Float 1062
Pi 1063
Point2f 68
UniformSampleDisk() 777

Figure 13.11: The mapping from 2D random samples to points on the disk implemented in `UniformSampleDisk()` distorts areas substantially. Each section of the disk here has equal area and represents $\frac{1}{8}$ of the unit square of uniform random samples in each direction. In general, we'd prefer a mapping that did a better job at mapping nearby (ξ_1, ξ_2) values to nearby points on the disk.

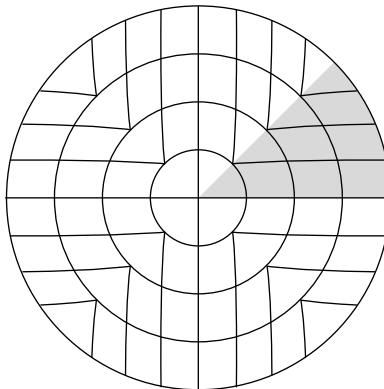


Figure 13.12: The concentric mapping maps squares to circles, giving a less distorted mapping than the first method shown for uniformly sampling points on the unit disk.

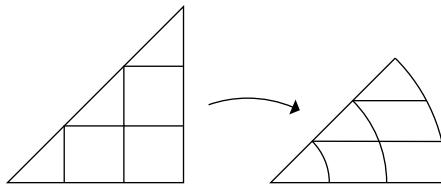


Figure 13.13: Triangular wedges of the square are mapped into (r, θ) pairs in pie-shaped slices of the circle.

to the unit disk by uniformly mapping concentric squares to concentric circles (Figure 13.12).

The mapping turns wedges of the square into slices of the disk. For example, points in the shaded area of the square in Figure 13.12 are mapped to (r, θ) by

$$\begin{aligned} r &= x \\ \theta &= \frac{y}{x} \pi. \end{aligned}$$

See Figure 13.13. The other two wedges are handled analogously.

(Sampling Function Definitions) +≡

```
Point2f ConcentricSampleDisk(const Point2f &u) {
    (Map uniform random numbers to  $[-1, 1]^2$  779)
    (Handle degeneracy at the origin 779)
    (Apply concentric mapping to point 779)
}
```

Point2f 68

```

⟨Map uniform random numbers to [−1, 1]²⟩ ≡ 778
    Point2f uOffset = 2.f * u - Vector2f(1, 1);

⟨Handle degeneracy at the origin⟩ ≡ 778
    if (uOffset.x == 0 && uOffset.y == 0)
        return Point2f(0, 0);

⟨Apply concentric mapping to point⟩ ≡ 778
    Float theta, r;
    if (std::abs(uOffset.x) > std::abs(uOffset.y)) {
        r = uOffset.x;
        theta = PiOver4 * (uOffset.y / uOffset.x);
    } else {
        r = uOffset.y;
        theta = PiOver2 - PiOver4 * (uOffset.x / uOffset.y);
    }
    return r * Point2f(std::cos(theta), std::sin(theta));

```

13.6.3 COSINE-WEIGHTED HEMISPHERE SAMPLING

As we will see in Section 13.10, it is often useful to sample from a distribution that has a shape similar to that of the integrand being estimated. For example, because the scattering equation weights the product of the BSDF and the incident radiance with a cosine term, it is useful to have a method that generates directions that are more likely to be close to the top of the hemisphere, where the cosine term has a large value, than the bottom, where the cosine term is small.

Mathematically, this means that we would like to sample directions ω from a PDF

$$p(\omega) \propto \cos \theta.$$

Normalizing as usual,

$$\begin{aligned} \int_{\mathbb{H}^2} p(\omega) d\omega &= 1 \\ \int_0^{2\pi} \int_0^{\frac{\pi}{2}} c \cos \theta \sin \theta d\theta d\phi &= 1 \\ c 2\pi \int_0^{\pi/2} \cos \theta \sin \theta d\theta &= 1 \\ c &= \frac{1}{\pi} \end{aligned}$$

so

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

Float 1062
PiOver2 1063
PiOver4 1063
Point2f 68
Vector2f 60

We could compute the marginal and conditional densities as before, but instead we can use a technique known as *Malley's method* to generate these cosine-weighted points. The idea behind Malley's method is that if we choose points uniformly from the unit disk and then generate directions by projecting the points on the disk up to the hemisphere above it, the resulting distribution of directions will have a cosine distribution (Figure 13.14).

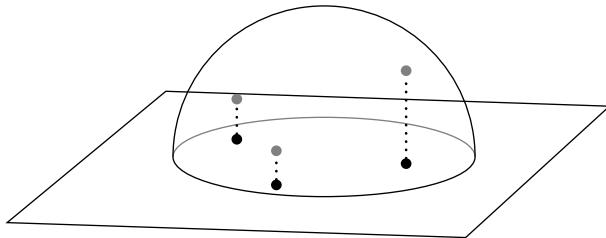


Figure 13.14: Mallery's Method. To sample direction vectors from a cosine-weighted distribution, uniformly sample points on the unit disk and project them up to the unit sphere.

Why does this work? Let (r, ϕ) be the polar coordinates of the point chosen on the disk (note that we're using ϕ instead of the usual θ here). From our calculations before, we know that the joint density $p(r, \phi) = r/\pi$ gives the density of a point sampled on the disk.

Now, we map this to the hemisphere. The vertical projection gives $\sin \theta = r$, which is easily seen from Figure 13.14. To complete the $(r, \phi) = (\sin \theta, \phi) \rightarrow (\theta, \phi)$ transformation, we need the determinant of the Jacobian

$$|J_T| = \begin{vmatrix} \cos \theta & 0 \\ 0 & 1 \end{vmatrix} = \cos \theta.$$

Therefore,

$$p(\theta, \phi) = |J_T| p(r, \phi) = \cos \theta \frac{r}{\pi} = (\cos \theta \sin \theta)/\pi,$$

which is exactly what we wanted! We have used the transformation method to prove that Mallery's method generates directions with a cosine-weighted distribution. Note that this technique works regardless of the method used to sample points from the circle, so we can use the earlier concentric mapping just as well as the simpler $(r, \theta) = (\sqrt{\xi_1}, 2\pi\xi_2)$ method.

(Sampling Inline Functions) +≡

```
inline Vector3f CosineSampleHemisphere(const Point2f &u) {
    Point2f d = ConcentricSampleDisk(u);
    Float z = std::sqrt(std::max((Float)0, 1 - d.x * d.x - d.y * d.y));
    return Vector3f(d.x, d.y, z);
}
```

Remember that all of the directional PDF evaluation routines in pbrt are defined with respect to solid angle, not spherical coordinates, so the PDF function returns a weight of $\cos \theta/\pi$.

(Sampling Inline Functions) +≡

```
inline Float CosineHemispherePdf(Float cosTheta) {
    return cosTheta * InvPi;
}
```

ConcentricSampleDisk() 778

Float 1062

InvPi 1063

Point2f 68

Vector3f 60

13.6.4 SAMPLING A CONE

For both area light sources based on Spheres as well as for the SpotLight, it's useful to be able to uniformly sample rays in a cone of directions. Such distributions are separable in (θ, ϕ) , with $p(\phi) = 1/(2\pi)$ and so we therefore need to derive a method to sample a direction θ uniformly over the cone of directions around a central direction up to the maximum angle of the beam, θ_{\max} . Incorporating the $\sin \theta$ term from the measure on the unit sphere from Equation (5.5),

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

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

{Sampling Function Definitions} +≡

```
Float UniformConePdf(Float cosThetaMax) {
    return 1 / (2 * Pi * (1 - cosThetaMax));
}
```

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

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

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

{Sampling Function Definitions} +≡

```
Vector3f UniformSampleCone(const Point2f &u, Float cosThetaMax) {
    Float cosTheta = ((Float)1 - u[0]) + u[0] * cosThetaMax;
    Float sinTheta = std::sqrt((Float)1 - cosTheta * cosTheta);
    Float phi = u[1] * 2 * Pi;
    return Vector3f(std::cos(phi) * sinTheta, std::sin(phi) * sinTheta,
                    cosTheta);
}
```

13.6.5 SAMPLING A TRIANGLE

Although uniformly sampling a triangle may seem like a simple task, it turns out to be more complex than the ones we've seen so far.⁹ To simplify the problem, we will assume we are sampling an isosceles right triangle of area $\frac{1}{2}$. The output of the sampling routine that we will derive will be barycentric coordinates, however, so the technique will actually

Float 1062
 Pi 1063
 Point2f 68
 SpotLight 721
 UniformSampleCone() 781
 Vector3f 60

9 It is possible to generate the right distribution in a triangle by sampling the enclosing parallelogram and reflecting samples on the wrong side of the diagonal back into the triangle. Although this technique is simpler than the one presented here, it is undesirable since it effectively folds the 2D uniform random samples back on top of each other—two samples that are very far away (e.g., $(.01, .01)$ and $(.99, .99)$) can map to the same point on the triangle. This thwarts variance reduction techniques like stratified sampling that generate sets of well-distributed (ξ_1, ξ_2) samples and expect that they will map to well-distributed points on the object being sampled; see Section 13.8.1 for further discussion.

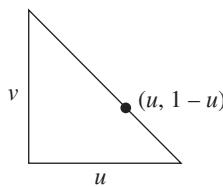


Figure 13.15: Sampling an Isosceles Right Triangle. Note that the equation of the hypotenuse is $v = 1 - u$.

work for any triangle despite this simplification. Figure 13.15 shows the shape to be sampled.

We will denote the two barycentric coordinates here by (u, v) . Since we are sampling with respect to area, we know that the PDF $p(u, v)$ must be a constant equal to the reciprocal of the shape's area, $\frac{1}{2}$, so $p(u, v) = 2$.

First, we find the marginal density $p(u)$:

$$p(u) = \int_0^{1-u} p(u, v) dv = 2 \int_0^{1-u} dv = 2(1-u),$$

and the conditional density $p(v|u)$:

$$p(v|u) = \frac{p(u, v)}{p(u)} = \frac{2}{2(1-u)} = \frac{1}{1-u}.$$

The CDFs are, as always, found by integration:

$$\begin{aligned} P(u) &= \int_0^u p(u') du' = 2u - u^2 \\ P(v) &= \int_0^v p(v'|u) dv' = \frac{v}{1-u}. \end{aligned}$$

Inverting these functions and assigning them to uniform random variables gives the final sampling strategy:

$$u = 1 - \sqrt{\xi_1}$$

$$v = \xi_2 \sqrt{\xi_1}.$$

Notice that the two variables in this case are *not* independent!

(Sampling Function Definitions) \equiv

```
Point2f UniformSampleTriangle(const Point2f &u) {
    Float su0 = std::sqrt(u[0]);
    return Point2f(1 - su0, u[1] * su0);
}
```

Float 1062

Point2f 68

We won't provide a PDF evaluation function for this sampling strategy since the PDF's value is just one over the triangle's area.

* 13.6.6 SAMPLING CAMERAS

Section 6.4.7 described the radiometry of the measurement that is performed by film in a realistic camera. For reference, the equation that gave joules arriving over an area of the sensor, Equation (6.7), is

$$J = \frac{1}{z^2} \int_{A_p} \int_{t_0}^{t_1} \int_{A_e} L_i(p, p', t') |\cos^4 \theta| dA_e dt' dA_p,$$

where the areas integrated over are the area of a pixel on the film, A_p , and an area over the rear lens element that bounds the lens's exit pupil, A_e . The integration over the pixel in the film is handled by the Integrator and the Sampler, so here we'll consider the estimate

$$\frac{1}{z^2} \int_{t_0}^{t_1} \int_{A_e} L_i(p, p', t') |\cos^4 \theta| dA_e dt'$$

for a fixed point on the film plane p .

Given a PDF for sampling a time and a PDF for sampling a point on the exit pupil, $p(A_e)$, we have the estimator

$$\frac{1}{z^2} \frac{L_i(p, p', t) |\cos^4 \theta|}{p(t) p(A_e)}.$$

For a uniformly sampled time value, $p(t) = 1/(t_1 - t_0)$. The weighting factor that should be applied to the incident radiance function L_i to compute an estimate of Equation (6.7) is then

$$\frac{(t_1 - t_0) |\cos^4 \theta|}{z^2 p(A_e)}.$$

Recall from Section 6.4.5 that points on the exit pupil were uniformly sampled within a 2D bounding box. Therefore, to compute $p(A_e)$, we just need one over the area of this bounding box. Conveniently, `RealisticCamera::SampleExitPupil()` returns this value, which `GenerateRay()` stores in `exitPupilBoundsArea`.

The computation of this weight is implemented in `(Return weighting for RealisticCamera ray)`; we can now see that if the `simpleWeighting` member variable is true, then the `RealisticCamera` computes a modified version of this term that only accounts for the $\cos^4 \theta$ term. While the value it computes isn't a useful physical quantity, this option gives images with pixel values that are closely related to the scene radiance (with vignetting), which is often convenient.

```
Camera::shutterClose 356
Camera::shutterOpen 356
Float 1062
Integrator 25
Ray::d 73
RealisticCamera 378
RealisticCamera:: GenerateRay()
394
RealisticCamera::LensRearZ()
381
RealisticCamera:: SampleExitPupil()
393
RealisticCamera:: simpleWeighting
379
Sampler 421
Vector3::Normalize() 66
```

(Return weighting for RealisticCamera ray) ≡ 394

```
    Float cosTheta = Normalize(rFilm.d).z;
    Float cos4Theta = (cosTheta * cosTheta) * (cosTheta * cosTheta);
    if (simpleWeighting)
        return cos4Theta;
    else
        return (shutterClose - shutterOpen) *
            (cos4Theta * exitPupilBoundsArea) / (LensRearZ() * LensRearZ());
```

13.6.7 PIECEWISE-CONSTANT 2D DISTRIBUTIONS

Our final example will show how to sample from discrete 2D distributions. We will consider the case of a 2D function defined over $(u, v) \in [0, 1]^2$ by a 2D array of $n_u \times n_v$ sample values. This case is particularly useful for generating samples from distributions defined by texture maps and environment maps.

Consider a 2D function $f(u, v)$ defined by a set of $n_u \times n_v$ values $f[u_i, v_j]$ where $u_i \in [0, 1, \dots, n_u - 1]$, $v_j \in [0, 1, \dots, n_v - 1]$, and $f[u_i, v_j]$ give the constant value of f over the range $[i/n_u, (i + 1)/n_u] \times [j/n_v, (j + 1)/n_v]$. Given continuous values (u, v) , we will use (\tilde{u}, \tilde{v}) to denote the corresponding discrete (u_i, v_j) indices, with $\tilde{u} = \lfloor n_u u \rfloor$ and $\tilde{v} = \lfloor n_v v \rfloor$ so that $f(u, v) = f[\tilde{u}, \tilde{v}]$.

Integrals of f are simple sums of $f[u_i, v_j]$ values, so that, for example, the integral of f over the domain is

$$I_f = \iint f(u, v) \, du \, dv = \frac{1}{n_u n_v} \sum_{i=0}^{n_u-1} \sum_{j=0}^{n_v-1} f[u_i, v_j].$$

Using the definition of the PDF and the integral of f , we can find f 's PDF,

$$p(u, v) = \frac{f(u, v)}{\iint f(u, v) \, du \, dv} = \frac{f[\tilde{u}, \tilde{v}]}{1/(n_u n_v) \sum_i \sum_j f[u_i, v_j]}.$$

Recalling Equation (13.11), the marginal density $p(v)$ can be computed as a sum of $f[u_i, v_j]$ values

$$p(v) = \int p(u, v) \, du = \frac{(1/n_u) \sum_i f[u_i, \tilde{v}]}{I_f}. \quad [13.13]$$

Because this function only depends on \tilde{v} , it is thus itself a piecewise constant 1D function, $p[\tilde{v}]$, defined by n_v values. The 1D sampling machinery from Section 13.3.1 can be applied to sampling from its distribution.

Given a v sample, the conditional density $p(u|v)$ is then

$$p(u|v) = \frac{p(u, v)}{p(v)} = \frac{f[\tilde{u}, \tilde{v}]/I_f}{p[\tilde{v}]}.$$

Note that, given a particular value of \tilde{v} , $p[\tilde{u}|\tilde{v}]$ is a piecewise-constant 1D function of \tilde{u} , that can be sampled with the usual 1D approach. There are n_v such distinct 1D conditional densities, one for each possible value of \tilde{v} .

Putting this all together, the `Distribution2D` structure provides functionality similar to `Distribution1D`, except that it generates samples from piecewise-constant 2D distributions.

```
(Sampling Declarations) +≡
  class Distribution2D {
    public:
      (Distribution2D Public Methods 786)
    private:
      (Distribution2D Private Data 785)
  };
```

The constructor has two tasks. First, it computes a 1D conditional sampling density $p[\tilde{u}|\tilde{v}]$ for each of the n_v individual \tilde{v} values using Equation (13.14). It then computes the marginal sampling density $p[\tilde{v}]$ with Equation (13.13).

```
(Sampling Function Definitions) +≡
  Distribution2D::Distribution2D(const Float *func, int nu, int nv) {
    for (int v = 0; v < nv; ++v) {
      (Compute conditional sampling distribution for  $\tilde{v}$  785)
    }
    (Compute marginal sampling distribution  $p[\tilde{v}]$  785)
  }
```

`Distribution1D` can directly compute the $p[\tilde{u}|\tilde{v}]$ distributions from a pointer to each of the n_v rows of n_u function values, since they're laid out linearly in memory. The I_f and $p[\tilde{v}]$ terms from Equation (13.14) don't need to be included in the values passed to `Distribution1D`, since they have the same value for all of the n_u values and are thus just a constant scale that doesn't affect the normalized distribution that `Distribution1D` computes.

```
(Compute conditional sampling distribution for  $\tilde{v}$  ) ≡ 785
  pConditionalV.emplace_back(new Distribution1D(&func[v * nu], nu));

(Distribution2D Private Data) ≡ 785
  std::vector<std::unique_ptr<Distribution1D>> pConditionalV;
```

Given the conditional densities for each \tilde{v} value, we can find the 1D marginal density for sampling each \tilde{v} value, $p[\tilde{v}]$. The `Distribution1D` class stores the integral of the piecewise-constant function it represents in its `funcInt` member variable, so it's just necessary to copy these values to the `marginalFunc` buffer so they're stored linearly in memory for the `Distribution1D` constructor.

```
(Compute marginal sampling distribution  $p[\tilde{v}]$  ) ≡ 785
  std::vector<Float> marginalFunc;
  for (int v = 0; v < nv; ++v)
    marginalFunc.push_back(pConditionalV[v]->funcInt);
  pMarginal.reset(new Distribution1D(&marginalFunc[0], nv));

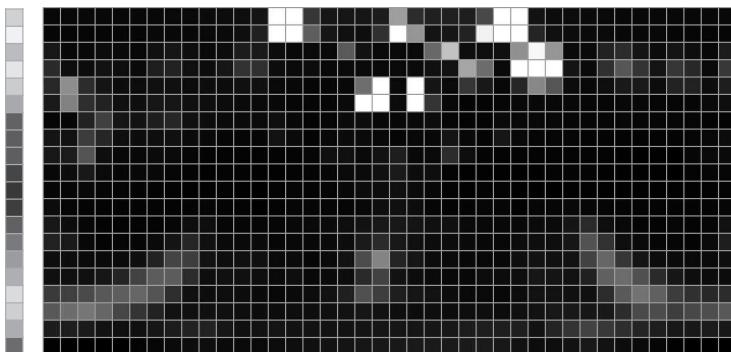
(Distribution2D Private Data) +≡ 785
  std::unique_ptr<Distribution1D> pMarginal;
```

`Distribution1D` 758
`Distribution1D::funcInt` 758
`Distribution2D` 785
`Distribution2D::pConditionalV` 785
`Distribution2D::pMarginal` 785
`Float` 1062

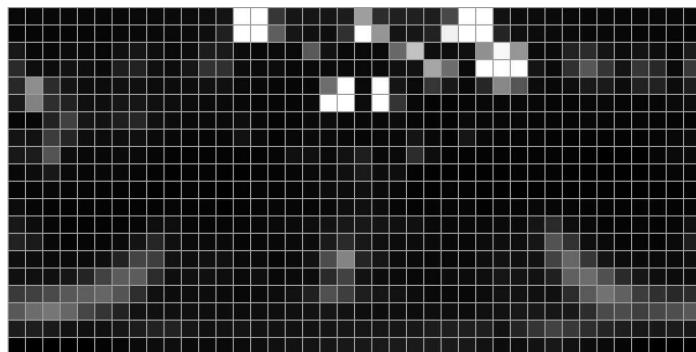
As described previously, in order to sample from the 2D distribution, first a sample is drawn from the $p[\tilde{v}]$ marginal distribution in order to find the v coordinate of the sample. The offset of the sampled function value gives the integer \tilde{v} value that determines



(a)



(b)



(c)

Figure 13.16: The Piecewise-Constant Sampling Distribution for a High-Dynamic-Range Environment Map. (a) The original environment map, (b) a low-resolution version of the marginal density function $p(\hat{v})$, (c) the conditional distributions for rows of the image. First the marginal 1D distribution (b) is used to select a v value, giving a row of the image to sample. Rows with bright pixels are more likely to be sampled. Then, given a row, a value u is sampled from that row's 1D distribution. (Grace Cathedral environment map courtesy of Paul Debevec.)

which of the precomputed conditional distributions should be used for sampling the u value. Figure 13.16 illustrates this idea using a low-resolution image as an example.

(Distribution2D Public Methods) ≡

```
Point2f SampleContinuous(const Point2f &u, Float *pdf) const {
    Float pdfs[2];
    int v;
    Float d1 = pMarginal->SampleContinuous(u[1], &pdfs[1], &v);
    Float d0 = pConditionalV[v]->SampleContinuous(u[0], &pdfs[0]);
    *pdf = pdfs[0] * pdfs[1];
    return Point2f(d0, d1);
}
```

785

```
Distribution1D::
    SampleContinuous()
759
```

```
Distribution2D::pConditionalV
785
```

```
Distribution2D::pMarginal
785
```

```
Float 1062
```

```
Point2f 68
```

The value of the PDF for a given sample value is computed as the product of the conditional and marginal PDFs for sampling it from the distribution.

```
(Distribution2D Public Methods) +≡
    Float Pdf(const Point2f &p) const {
        int iu = Clamp(int(p[0] * pConditionalV[0]->Count()),
                      0, pConditionalV[0]->Count() - 1);
        int iv = Clamp(int(p[1] * pMarginal->Count()),
                      0, pMarginal->Count() - 1);
        return pConditionalV[iv]->func[iu] / pMarginal->funcInt;
    }
```

785

13.7 RUSSIAN ROULETTE AND SPLITTING

The *efficiency* of an estimator F is defined as

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

where $V[F]$ is its variance and $T[F]$ is the running time to compute its value. According to this metric, an estimator F_1 is more efficient than F_2 if it takes less time to produce the same variance, or if it produces less variance in the same amount of time. The next few sections discuss a number of techniques for improving the efficiency of Monte Carlo.

Russian roulette and splitting are two related techniques that can improve the efficiency of Monte Carlo estimates by increasing the likelihood that each sample will have a significant contribution to the result. Russian roulette addresses the problem of samples that are expensive to evaluate but make a small contribution to the final result, and splitting is a technique that makes it possible to place more samples in important dimensions of the integrand.

As an example to motivate Russian roulette, consider the problem of estimating the direct lighting integral, which gives reflected radiance at a point due to radiance from direct illumination from the light sources in the scene, L_d :

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

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

$$\frac{1}{2} \sum_{i=1}^2 \frac{f_r(p, \omega_o, \omega_i) L_d(p, \omega_i) |\cos \theta_i|}{p(\omega_i)}.$$

Most of the computational expense of evaluating each term of the sum comes from tracing a shadow ray from the point p to see whether the light source is occluded as seen from p .

For all of the directions ω_i where the integrand's value is 0 because $f_r(p, \omega_o, \omega_i)$ is 0 for that direction, we should obviously skip the work of tracing the shadow ray, since tracing it won't change the final value computed. Russian roulette makes it possible to

Clamp() 1062
 Distribution1D::Count() 758
 Distribution1D::func 758
 Distribution1D::funcInt 758
 Distribution2D::pConditionalV 785
 Float 1062
 Point2f 68

also skip tracing rays when the integrand's value is very low but not necessarily 0, while still computing the correct value on average. For example, we might want to avoid tracing rays when $f_r(p, \omega_o, \omega_i)$ is small or when ω_i is close to the horizon and thus $|\cos \theta_i|$ is small. These samples just can't be ignored completely, however, since the estimator would then consistently underestimate the correct result.

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

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

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

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

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

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

13.7.1 SPLITTING

While Russian roulette reduces the effort spent evaluating unimportant samples, splitting increases the number of samples taken in order to improve efficiency. Consider again the problem of computing reflection due only to direct illumination. Ignoring pixel filtering, this problem can be written as a double integral over the area of the pixel A and over the sphere of directions S^2 at the visible points on surfaces at each (x, y) pixel position:

$$\int_A \int_{S^2} L_d(x, y, \omega) \, dx \, dy \, d\omega,$$

where $L_d(x, y, \omega)$ denotes the exitant radiance at the object visible at the position (x, y) on the image due to incident radiance from the direction ω .

The natural way to estimate the integral is to generate N samples and apply the Monte Carlo estimator, where each sample consists of an (x, y) image position and a direction

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

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

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

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

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

The purpose of the `Sampler::Request1DArray()` and `Sampler::Request2DArray()` methods defined in Section 7.2.2 can now be seen in the light that they make it possible for users of Samplers to apply splitting to some dimensions of the integrand.

13.8 CAREFUL SAMPLE PLACEMENT

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

13.8.1 STRATIFIED SAMPLING

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

```
Sampler 421
Sampler::Request1DArray()
 423
Sampler::Request2DArray()
 423
```

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

To draw samples from Λ , we will draw n_i samples from each Λ_i , according to densities p_i inside each stratum. A simple example is supersampling a pixel. With stratified sampling,

the area around a pixel is divided into a $k \times k$ grid, and a sample is drawn from each grid cell. This is better than taking k^2 random samples, since the sample locations are less likely to clump together. Here we will show why this technique reduces variance.

Within a single stratum Λ_i , the Monte Carlo estimate is

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

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

The true value of the integrand in stratum i is

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

and the variance in this stratum is

$$\sigma_i^2 = \frac{1}{v_i} \int_{\Lambda_i} (f(x) - \mu_i)^2 dx.$$

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

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

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

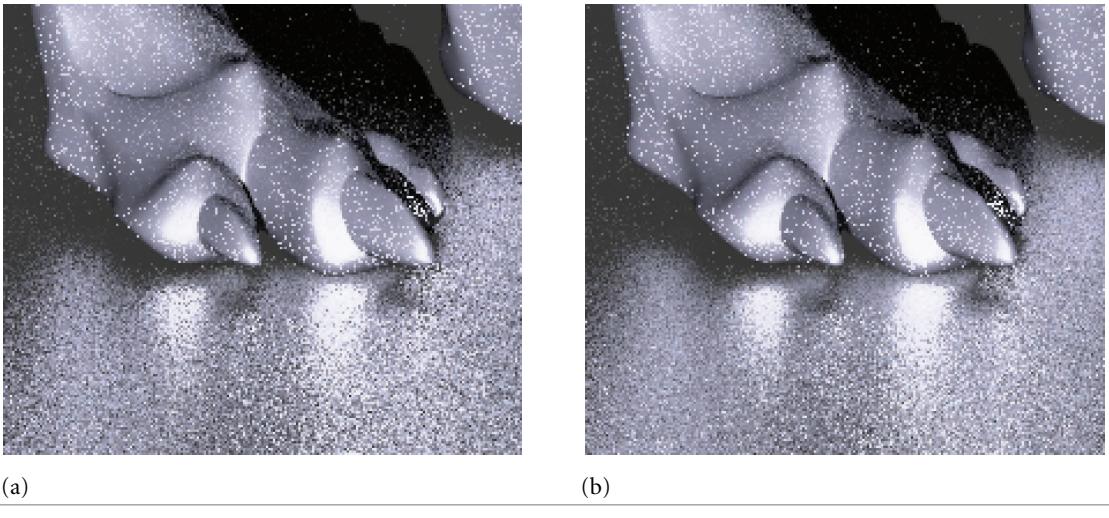
$$V[F_N] = \frac{1}{N} \sum v_i \sigma_i^2.$$

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

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

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

There are two things to notice about this expression. First, we know that the right-hand sum must be nonnegative, since variance is always nonnegative. Second, it demonstrates



(a)

(b)

Figure 13.17: Variance is higher and the image noisier (a) when random sampling is used to compute the effect of glossy reflection than (b) when a stratified distribution of sample directions is used instead. (Compare the edges of the highlights on the ground, for example.)

that stratified sampling can never increase variance. In fact, stratification always reduces variance unless the right-hand sum is exactly 0. It can only be 0 when the function f has the same mean over each stratum Λ_i . In fact, for stratified sampling to work best, we would like to maximize the right-hand sum, so it is best to make the strata have means that are as unequal as possible. This explains why *compact* strata are desirable if one does not know anything about the function f . If the strata are wide, they will contain more variation and will have μ_i closer to the true mean Q .

Figure 13.17 shows the effect of using stratified sampling versus a uniform random distribution for sampling ray directions for glossy reflection. There is a reasonable reduction in variance at essentially no cost in running time.

The main downside of stratified sampling is that it suffers from the same “curse of dimensionality” as standard numerical quadrature. Full stratification in D dimensions with S strata per dimension requires S^D samples, which quickly becomes prohibitive. Fortunately, it is often possible to stratify some of the dimensions independently and then randomly associate samples from different dimensions, as was done in Section 7.3. Choosing which dimensions are stratified should be done in a way that stratifies dimensions that tend to be most highly correlated in their effect on the value of the integrand (Owen 1998). For example, for the direct lighting example in Section 13.7.1, it is far more effective to stratify the (x, y) pixel positions and to stratify the (θ, ϕ) ray direction—stratifying (x, θ) and (y, ϕ) would almost certainly be ineffective.

Another solution to the curse of dimensionality that has many of the same advantages of stratification is to use Latin hypercube sampling (also introduced in Section 7.3), which can generate any number of samples independent of the number of dimensions. Unfortunately, Latin hypercube sampling isn’t as effective as stratified sampling at reducing

variance, especially as the number of samples taken becomes large. Nevertheless, Latin hypercube sampling is provably no worse than uniform random sampling and is often much better.

13.8.2 QUASI MONTE CARLO

The low-discrepancy sampling techniques introduced in Chapter 7 are the foundation of a branch of Monte Carlo called *quasi Monte Carlo*. The key component of quasi-Monte Carlo techniques is that they replace the pseudo-random numbers used in standard Monte Carlo with low-discrepancy point sets generated by carefully designed deterministic algorithms.

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

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

13.8.3 WARPING SAMPLES AND DISTORTION

When applying stratified sampling or low-discrepancy sampling to problems like choosing points on light sources for integration for area lighting, pbrt generates a set of samples (u_1, u_2) over the domain $[0, 1]^2$ and then uses algorithms based on the transformation methods introduced in Sections 13.5 and 13.6 to map these samples to points on the light source. Implicit in this process is the expectation that the transformation to points on the light source will generally preserve the stratification properties of the samples from $[0, 1]^2$ —in other words, nearby samples should map to nearby positions on the surface of the light, and faraway samples should map to far-apart positions on the light. If the mapping does not preserve this property, then the benefits of stratification are lost.

Shirley’s square-to-circle mapping (Figure 13.13) preserves stratification more effectively than the straightforward mapping (Figure 13.12), which has less compact strata away from the center. This issue also explains why low-discrepancy sequences are generally more effective than stratified patterns in practice: they are more robust with respect to preserving their good distribution properties after being transformed to other domains. Figure 13.18 shows what happens when a set of 16 well-distributed sample points are transformed to be points on a skinny quadrilateral by scaling them to cover its surface;

`Integrator` 25

`Sampler` 421

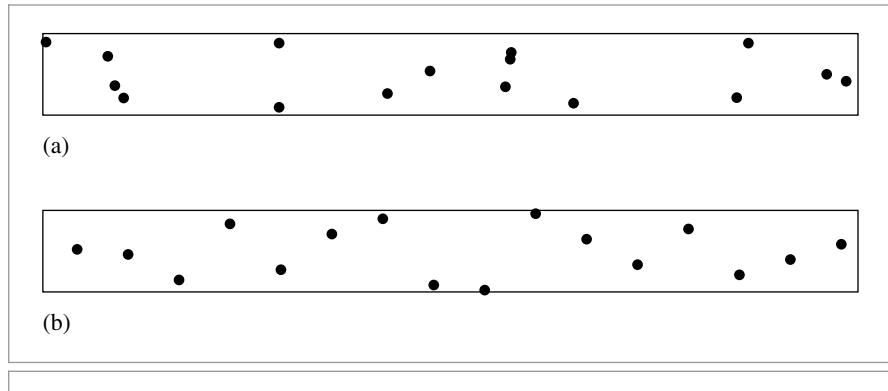


Figure 13.18: (a) Transforming a 4×4 stratified sampling pattern to points on a long and thin quadrilateral light source effectively gives less than 16 well-distributed samples; stratification in the vertical direction is not helpful. (b) Samples from a $(0, 2)$ -sequence remain well distributed even after this transformation.

samples from a $(0, 2)$ -sequence remain well distributed, but samples from a stratified pattern fare much less well.

13.9 BIAS

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

$$\beta = E[F] - \int f(x) dx$$

is the amount of bias.

Kalos and Whitlock (1986, pp. 36–37) gave the following example of how bias can sometimes be desirable. Consider the problem of computing an estimate of the mean value of a uniform distribution $X_i \sim p$ over the interval from 0 to 1. One could use the estimator

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

or one could use the biased estimator

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

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

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

so it is biased, although its variance is $O(N^{-2})$, which is much better.

The pixel reconstruction method described in Section 7.8 can also be seen as a biased estimator. Considering pixel reconstruction as a Monte Carlo estimation problem, we'd like to compute an estimate of

$$I(x, y) = \iint f(x - x', y - y') L(x', y') dx' dy',$$

where $I(x, y)$ is a final pixel value, $f(x, y)$ is the pixel filter function (which we assume here to be normalized to integrate to 1), and $L(x, y)$ is the image radiance function.

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

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

This gives a different result from that of the pixel filtering equation we used previously, Equation (7.12), which was

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

Yet, the biased estimator is preferable in practice because it gives a result with less variance. For example, if all radiance values $L(x_i, y_i)$ have a value of 1, the biased estimator will always reconstruct an image where all pixel values are exactly 1—clearly a desirable property. However, the unbiased estimator will reconstruct pixel values that are not all 1, since the sum

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

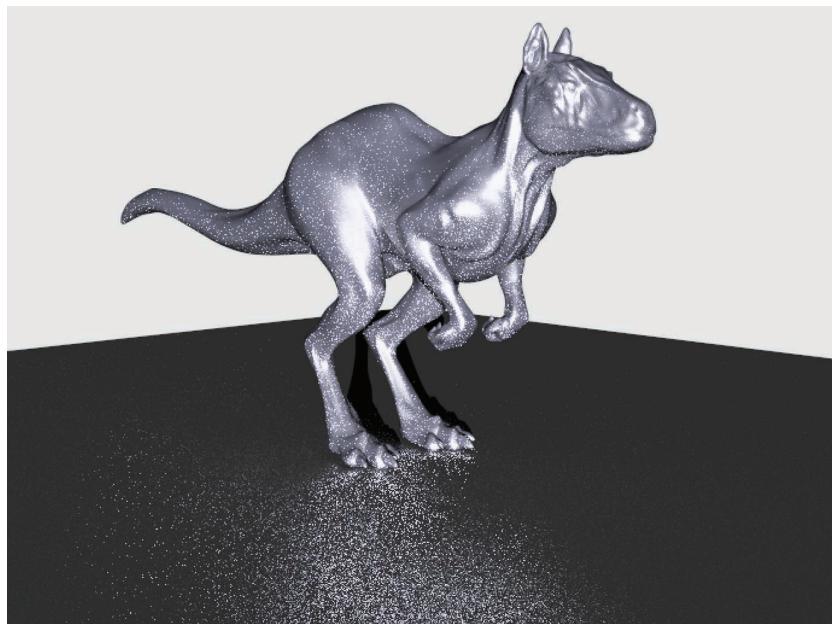
will generally not be equal to p_c and thus will have a different value due to variation in the filter function depending on the particular (x_i, y_i) sample positions used for the pixel. Thus, the variance due to this effect leads to an undesirable result in the final image. Even for more complex images, the variance that would be introduced by the unbiased estimator is a more objectionable artifact than the bias from Equation (7.12).

13.10 IMPORTANCE SAMPLING

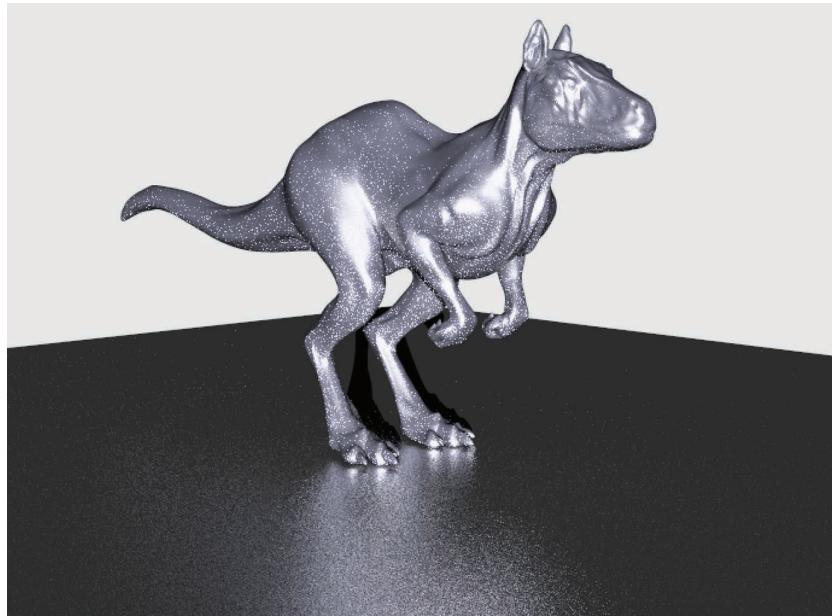
Importance sampling is a powerful variance reduction technique that exploits the fact that the Monte Carlo estimator

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

converges more quickly if the samples are taken from a distribution $p(x)$ that is similar to the function $f(x)$ in the integrand. The basic idea is that by concentrating work where the value of the integrand is relatively high, an accurate estimate is computed more efficiently (Figure 13.19).



(a)



(b)

Figure 13.19: (a) Using a stratified uniform distribution of rays over the hemisphere gives an image with much more variance than (b) applying importance sampling and choosing stratified rays from a distribution based on the BRDF.

For example, suppose we are evaluating the scattering equation, Equation (5.9). Consider what happens when this integral is estimated; if a direction is randomly sampled that is nearly perpendicular to the surface normal, the cosine term will be close to 0. All the expense of evaluating the BSDF and tracing a ray to find the incoming radiance at that sample location will be essentially wasted, as the contribution to the final result will be minuscule. In general, we would be better served if we sampled directions in a way that reduced the likelihood of choosing directions near the horizon. More generally, if directions are sampled from distributions that match other factors of the integrand (the BSDF, the incoming illumination distribution, etc.), efficiency is similarly improved.

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

$$c = \frac{1}{\int f(x) dx}.$$

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

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

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

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

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

to compute the estimate of $\int f(x) dx$ where

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

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

necessary to balance out these extremes to reduce variance enough to get a result close to the actual value, 1.

Fortunately, it's not too hard to find good sampling distributions for importance sampling for many integration problems in graphics. As we implement Integrators in the next three chapters, we will derive a variety of sampling distributions for BSDFs, lights, cameras, and participating media. In many cases, the integrand is the product of more than one function. It can be difficult to construct a PDF that is similar to the complete product, but finding one that is similar to one of the multiplicands is still helpful.

In practice, importance sampling is one of the most frequently used variance reduction techniques in rendering, since it is easy to apply and is very effective when good sampling distributions are used.

13.10.1 MULTIPLE IMPORTANCE SAMPLING

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

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

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

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

Consider a near-mirror BRDF illuminated by an area light where L_d 's distribution is used to draw samples. Because the BRDF is almost a mirror, the value of the integrand will be close to 0 at all ω_i directions except those around the perfect specular reflection direction. This means that almost all of the directions sampled by L_d will have 0 contribution, and variance will be quite high. Even worse, as the light source grows large and a larger set of directions is potentially sampled, the value of the PDF decreases, so for the rare directions where the BRDF is non-0 for the sampled direction we will have a large integrand value being divided by a small PDF value. While sampling from the BRDF's distribution would be a much better approach to this particular case, for diffuse or glossy BRDFs and small light sources, sampling from the BRDF's distribution can similarly lead to much higher variance than sampling from the light's distribution.

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

Multiple importance sampling (MIS) addresses exactly this issue, with a simple and easy-to-implement technique. The basic idea is that, when estimating an integral, we should draw samples from multiple sampling distributions, chosen in the hope that at least one

of them will match the shape of the integrand reasonably well, even if we don't know which one this will be. MIS provides a method to weight the samples from each technique that can eliminate large variance spikes due to mismatches between the integrand's value and the sampling density. Specialized sampling routines that only account for unusual special cases are even encouraged, as they reduce variance when those cases occur, with relatively little cost in general. See Figure 14.13 in Chapter 14 to see the reduction in variance from using MIS to compute reflection from direct illumination compared to sampling either just the BSDF or the light's distribution by itself.

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

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

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

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

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

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

Consider the effect of this term for the case where a sample X has been drawn from the p_f distribution at a point where the value $p_f(X)$ is relatively low. Assuming that p_f is a good match for the shape of $f(x)$, then the value of $f(X)$ will also be relatively low. But suppose that $g(X)$ has a relatively high value. The standard importance sampling estimate

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

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

With the balance heuristic, the contribution of X will be

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

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

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

```
{Sampling Inline Functions} +≡
    inline Float BalanceHeuristic(int nf, Float fPdf, int ng, Float gPdf) {
        return (nf * fPdf) / (nf * fPdf + ng * gPdf);
    }
```

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

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

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

```
{Sampling Inline Functions} +≡
    inline Float PowerHeuristic(int nf, Float fPdf, int ng, Float gPdf) {
        Float f = nf * fPdf, g = ng * gPdf;
        return (f * f) / (f * f + g * g);
    }
```

FURTHER READING

Many books have been written on Monte Carlo integration. Hammersley and Handscomb (1964), Spanier and Gelbard (1969), and Kalos and Whitlock (1986) are classic references. More recent books on the topic include those by Fishman (1996) and Liu (2001). Chib and Greenberg (1995) have written an approachable but rigorous introduction to the Metropolis algorithm. The Monte Carlo and Quasi Monte Carlo Web site is a useful gateway to recent work in the field (www.mcqmc.org).

Good general references about Monte Carlo and its application to computer graphics are the theses by Lafourche (1996) and Veach (1997). Dutré's *Global Illumination Compendium* (2003) also has much useful information related to this topic. The course notes from the Monte Carlo ray-tracing course at SIGGRAPH have a wealth of practical information (Jensen et al. 2001a, 2003).

The square to disk mapping was described by Shirley and Chiu (1997). The implementation here benefits by observations by Cline and "franz" that the logic could be simplified considerably from the original algorithm (Shirley 2011). Marques et al. (2013) note that well-distributed samples on $[0, 1]^2$ still suffer some distortion when they are mapped to the sphere of directions and show how to generate low-discrepancy samples on the unit sphere.

Steigleder and McCool (2003) described an alternative to the multidimensional sampling approach from Section 13.6.7: they linearized 2D and higher dimensional domains into 1D using a Hilbert curve and then sampled using 1D samples over the 1D domain. This leads to a simpler implementation that still maintains desirable stratification properties of the sampling distribution, thanks to the spatial coherence preserving properties of the Hilbert curve.

Lawrence et al. (2005) described an adaptive representation for CDFs, where the CDF is approximated with a piecewise linear function with fewer, but irregularly spaced, vertices compared to the complete CDF. This approach can substantially reduce storage requirements and improve lookup efficiency, taking advantage of the fact that large ranges of the CDF may be efficiently approximated with a single linear function.

Cline et al. (2009) observed that the time spent in binary searches for sampling from precomputed distributions (like `Distribution1D` does) can take a meaningful amount of execution time. (Indeed, `pbrt` spends nearly 7% of the time when rendering the car scene lit by an `InfiniteAreaLight` in the `Distribution1D::SampleContinuous()` method, which is used by `InfiniteAreaLight::Sample_Li()`.) They presented two improved methods for doing this sort of search: the first stores a lookup table with n integer values, indexed by $\lfloor n\xi \rfloor$, which gives the first entry in the CDF array that is less than or equal to ξ . Starting a linear search from this offset in the CDF array can be much more efficient than a complete binary search over the entire array. They also presented a method based on approximating the inverse CDF as a piecewise linear function of ξ and thus enabling constant-time lookups at a cost of some accuracy (and thus some additional variance).

The *alias method* is a technique that makes it possible to sample from discrete distributions in $O(1)$ time (Walker 1974, 1977); this is much better than the $O(\log n)$ of the `Distribution1D` class when it is used for sampling discrete distributions. The downside of this approach is that it does not preserve sample stratification. See Schwarz's writeup (2011) for information about implementing this approach well.

Arithmetic coding offers another interesting way to approach sampling from distributions (MacKay 2003, p. 118; Piponi 2012). If we have a discrete set of probabilities we'd like to generate samples from, one way to approach the problem is to encode the CDF as a binary tree where each node splits the $[0, 1]$ interval at some point and where, given a random sample ξ , we determine which sample value it corresponds to by traversing the tree until we reach the leaf node for its sample value. Ideally, we'd like to have leaf nodes that represent higher probabilities be higher up in the tree, so that it takes fewer traversal steps to find them (and thus, those more frequently generated samples are found more quickly). Looking at the problem from this perspective, it can be shown that the optimal structure of such a tree is given by Huffman coding, which is normally used for compression.

Mitchell (1996b) wrote a paper that investigates the effects of stratification for integration problems in graphics (including the 2D problem of pixel antialiasing). In particular, he investigated the connection between the complexity of the function being integrated and the effect of stratification. In general, the smoother or simpler the function, the more stratification helps: for pixels with smooth variation over their areas or with just a few edges passing through them, stratification helps substantially, but as the complexity in a pixel is increased, the gain from stratification is reduced. Nevertheless, because stratification never increases variance, there's no reason not to do it.

Starting with Durand et al.'s work (2005), a number of researchers have approached the analysis of light transport and variance from Monte Carlo integration for rendering using Fourier analysis. See Pilleboue et al.'s paper (2015) for recent work in this area, including references to previous work. Among other results, they show that Poisson disk patterns

`Distribution1D` 758

`Distribution1D::`
`SampleContinuous()`
`759`

`InfiniteAreaLight` 737

`InfiniteAreaLight::`
`Sample_Li()`
`849`

give higher variance than simple jittered patterns. They found that the blue noise pattern of de Goes et al. (2012) was fairly effective. Other work in this area includes the paper by Subr and Kautz (2013).

Multiple importance sampling was developed by Veach and Guibas (Veach and Guibas 1995; Veach 1997). Normally, a pre-determined number of samples are taken using each sampling technique; see Pajot et al. (2011) and Lu et al. (2013) for approaches to adaptively distributing the samples over strategies in an effort to reduce variance by choosing those that are the best match to the integrand.

EXERCISES

- ② 13.1 Write a program that compares Monte Carlo and one or more alternative numerical integration techniques. Structure this program so that it is easy to replace the particular function being integrated. Verify that the different techniques compute the same result (given a sufficient number of samples for each of them). Modify your program so that it draws samples from distributions other than the uniform distribution for the Monte Carlo estimate, and verify that it still computes the correct result when the correct estimator, Equation (13.3), is used. (Make sure that any alternative distributions you use have nonzero probability of choosing any value of x where $f(x) > 0$.)
- ① 13.2 Write a program that computes Monte Carlo estimates of the integral of a given function. Compute an estimate of the variance of the estimates by taking a series of trials and using Equation (13.2) to compute variance. Demonstrate numerically that variance decreases at a rate of $O(\sqrt{n})$.
- ① 13.3 The depth-of-field code for the `ProjectiveCamera` in Section 6.2.3 uses the `ConcentricSampleDisk()` function to generate samples on the circular lens, since this function gives less distortion than `UniformSampleDisk()`. Try replacing it with `UniformSampleDisk()`, and measure the difference in image quality. For example, you might want to compare the error in images from using each approach and a relatively low number of samples to a highly sampled reference image.

Does `ConcentricSampleDisk()` in fact give less error in practice? Does it make a difference if a relatively simple scene is being rendered versus a very complex scene?

- ② 13.4 Modify the `Distribution1D` implementation to use the adaptive CDF representation described by Lawrence et al. (2005), and experiment with how much more compact the CDF representation can be made without causing image artifacts. (Good test scenes include those that use `InfiniteAreaLights`, which use the `Distribution2D` and, thus, `Distribution1D` for sampling.) Can you measure an improvement in rendering speed due to more efficient searches through the approximated CDF?
- ③ 13.5 One useful technique not discussed in this chapter is the idea of adaptive density distribution functions that dynamically change the sampling distribution

`ConcentricSampleDisk()` 778
`Distribution1D` 758
`Distribution2D` 785
`InfiniteAreaLight` 737
`ProjectiveCamera` 358
`UniformSampleDisk()` 777

as samples are taken and information is available about the integrand's actual distribution as a result of evaluating the values of these samples. The standard Monte Carlo estimator can be written to work with a nonuniform distribution of random numbers used in a transformation method to generate samples X_i ,

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

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

Investigate data structures and algorithms that support such sampling approaches and choose a sampling problem in pbrt to apply them to. Measure how well this approach works for the problem you selected. One difficulty with methods like this is that different parts of the sampling domain will be the most effective at different times in different parts of the scene. For example, trying to adaptively change the sampling density of points over the surface of an area light source has to contend with the fact that, at different parts of the scene, different parts of the area light may be visible and thus be the important areas. You may find it useful to read Cline et al.'s paper (2008) on this topic.

This page intentionally left blank



CHAPTER FOURTEEN

14 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 two different 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).

Recall the scattering equation from Section 5.6.1; its value can be estimated with Monte Carlo:

$$\begin{aligned} L_o(p, \omega_o) &= \int_{S^2} f(p, \omega_o, \omega_i) L_i(p, \omega_i) |\cos \theta_i| d\omega_i \\ &\approx \frac{1}{N} \sum_{j=1}^N \frac{f(p, \omega_o, \omega_j) L_i(p, \omega_j) |\cos \theta_j|}{p(\omega_j)}, \end{aligned}$$

with directions ω_j sampled from a distribution with respect to solid angle that has PDF $p(\omega_j)$. In practice, we'll want to take some samples from a distribution that approximates the BSDF, some from a distribution that approximates the incident radiance from light sources, and then weight the samples with multiple importance sampling.

The next two sections derive methods for sampling from BSDFs and light sources. After these sampling methods have been defined, the `DirectLightingIntegrator` and the `PathIntegrator` will be introduced. Both find light-carrying paths starting from the camera, accounting for scattering from shapes' surfaces. Chapter 15 will extend this approach to scattering from participating media as well, and Chapter 16 will introduce bidirectional methods for constructing light-carrying paths starting from both the camera and from light sources.

14.1 SAMPLING REFLECTION FUNCTIONS

The `BxDF::Sample_f()` method chooses a direction according to a distribution that is similar to its corresponding scattering function. In Section 8.2, this method was used for finding reflected and transmitted rays from perfectly specular surfaces; later in this section, we will show how that sampling process is a special case of the sampling techniques we'll now implement for other types of BSDFs.

`BxDF::Sample_f()` takes two sample values in the range $[0, 1]^2$ that are intended to be used by an inversion method-based sampling algorithm (recall Section 13.3.1). The routine calling it can use stratified or low-discrepancy sampling techniques to generate these sample values, thus making it possible for the sampled directions themselves to be well distributed. Other sampling methods like rejection sampling could in theory be supported by passing a `Sampler` instance, though this is not done in `pbrt` as stratified sampling of a distribution that is similar to the BSDF generally produces superior results.

This method returns the value of the BSDF for the chosen direction as well as the sampled direction in `*wi` and the value of $p(\omega_i)$ in `*pdf`. The PDF value returned should be measured with respect to solid angle, and both the outgoing direction ω_o and the sampled incident direction ω_i should be in the standard reflection coordinate system (see section “Geometric Setting,” page 509).

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

(BxDF Method Definitions) +≡

```
Spectrum BxDF::Sample_f(const Vector3f &wo, Vector3f *wi,
    const Point2f &u, Float *pdf, BxDFType *sampledType) const {
    (Cosine-sample the hemisphere, flipping the direction if necessary 806)
    *pdf = Pdf(wo, *wi);
    return f(wo, *wi);
}
```

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

(Cosine-sample the hemisphere, flipping the direction if necessary) ≡

806, 814

```
*wi = CosineSampleHemisphere(u);
if (wo.z < 0) wi->z *= -1;
```

While `BxDF::Sample_f()` returns the value of the PDF for the direction it chose, the `BxDF::Pdf()` method returns the value of the PDF for a given pair of directions. This

BxDF 513
`BxDF::f()` 514
`BxDF::Pdf()` 807
`BxDF::Sample_f()` 806
`BxDFType` 513
`CosineSampleHemisphere()` 780
`Float` 1062
`Point2f` 68
`Sampler` 421
`Spectrum` 315
`Vector3f` 60

method is useful for multiple importance sampling, where it is necessary to be able to find one sampling distribution's PDF for directions sampled from other distributions. It is crucial that any BxDF implementation that overrides the `BxDF::Sample_f()` method also override the `BxDF::Pdf()` method so that the two return consistent results.¹

To actually evaluate the PDF for the cosine-weighted sampling method (which we showed earlier was $p(\omega) = \cos \theta / \pi$), it is first necessary to check that ω_o and ω_i lie on the same side of the surface; if not, the sampling probability is 0. Otherwise, the method computes $|\mathbf{n} \cdot \omega_i|$. One potential pitfall with this method is that the order of the ω_o and ω_i arguments is significant. For the cosine-weighted distribution, $p(\omega_o) \neq p(\omega_i)$ in general. Code that calls this method must be careful to use the correct argument ordering.

```
(BxDF Method Definitions) +≡
    Float BxDF::Pdf(const Vector3f &wo, const Vector3f &wi) const {
        return SameHemisphere(wo, wi) ? AbsCosTheta(wi) * InvPi : 0;
    }

(BSDF Inline Functions) +≡
    inline bool SameHemisphere(const Vector3f &w, const Vector3f &wp) {
        return w.z * wp.z > 0;
    }
```

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

14.1.1 MICROFACET BXDFs

The microfacet-based reflection models defined in Section 8.4 were based on a distribution of microfacets $D(\omega_h)$ where each microfacet exhibited perfect specular reflection and/or transmission. Because the $D(\omega_h)$ function is primarily responsible for the overall shape of the Torrance–Sparrow BSDF (Section 8.4.4), approaches based on sampling from its distribution are fairly effective. With this approach, first a particular microfacet orientation is sampled from the microfacet distribution, and then the incident direction is found using the specular reflection or transmission formula.

Therefore, `MicrofacetDistribution` implementations must implement a method for sampling from their distribution of normal vectors.

<code>AbsCosTheta()</code> 510 <code>BxDF</code> 513 <code>BxDF::Pdf()</code> 807 <code>BxDF::Sample_f()</code> 806 <code>Float</code> 1062 <code>InvPi</code> 1063 <code>MicrofacetDistribution</code> 537 <code>Point2f</code> 68 <code>SameHemisphere()</code> 807 <code>Vector3f</code> 60	537
---	------------

```
(MicrofacetDistribution Public Methods) +≡
    virtual Vector3f Sample_wh(const Vector3f &wo,
                                const Point2f &u) const = 0;
```

The classic approach to sampling a microfacet orientation is to sample from $D(\omega_h)$ directly. We'll start by showing the derivation of this approach for the isotropic Beckmann distribution but will then describe a more effective sampling method that samples from the distribution of visible microfacets from a given outgoing direction, which can be quite different from the overall distribution.

¹ The unit tests in the `src/tests` directory of pbrt source code distribution include a `bsdf.cpp` test (not discussed in the book) that implements a χ^2 statistical hypothesis test. This test checks the consistency of a BxDF's `BxDF::Sample_f()` and `BxDF::Pdf()` methods and can be helpful to validate sampling routines when implementing new types of BxDFs.

The `MicrofacetDistribution` base class stores a Boolean value that determines which sampling technique will be used. In practice, the one based on sampling visible microfacet area is much more effective than the one based on sampling the overall distribution, so it isn't possible to select between the two strategies in pbrt scene description files; the option to sample the overall distribution is only available for tests and comparisons.

(MicrofacetDistribution Protected Methods) \equiv

```
MicrofacetDistribution(bool sampleVisibleArea)
: sampleVisibleArea(sampleVisibleArea) { }
```

(MicrofacetDistribution Protected Data) \equiv

```
const bool sampleVisibleArea;
```

The `BeckmannDistribution`'s `Sample_wh()` method's implementation uses this value to determine which sampling technique to use.

(MicrofacetDistribution Method Definitions) $+ \equiv$

```
Vector3f BeckmannDistribution::Sample_wh(const Vector3f &wo,
                                         const Point2f &u) const {
    if (!sampleVisibleArea) {
        (Sample full distribution of normals for Beckmann distribution 808)
    } else {
        (Sample visible area of normals for Beckmann distribution)
    }
}
```

The sampling method for the Beckmann–Spizzichino distribution's full distribution of normals returns angles $\tan^2 \theta$ and ϕ in spherical coordinates, which in turn are converted to a normalized direction vector `wh`.

(Sample full distribution of normals for Beckmann distribution) \equiv

```
808
(Compute tan2 θ and φ for Beckmann distribution sample 809)
(Map sampled Beckmann angles to normal direction wh 809)
return wh;
```

The isotropic Beckmann–Spizzichino distribution was defined in Equation (8.10). To derive a sampling method, we'll consider it expressed in spherical coordinates. As an isotropic distribution, it is independent of ϕ , and so the PDF for this distribution, $p_h(\theta, \phi)$, is separable into $p_h(\theta)$ and $p_h(\phi)$.

$p_h(\phi)$ is constant with a value of $1/(2\pi)$, and thus ϕ values can be sampled by

$$\phi = 2\pi\xi.$$

For $p(\theta_h)$, we have

$$p(\theta_h) = \frac{e^{-\tan^2 \theta_h/\alpha^2}}{\pi \alpha^2 \cos^4 \theta_h}, \quad [14.1]$$

where α is the roughness coefficient. We can apply the inversion method to find how to sample a direction θ' from this distribution given a uniform random number ξ . First,

MicrofacetDistribution 537
 MicrofacetDistribution::
 sampleVisibleArea
 808
 Point2f 68
 Vector3f 60

taking the PDF from Equation (14.1), and integrating to find the CDF, we have

$$\begin{aligned} P(\theta') &= \int_0^{\theta'} \frac{e^{-\tan^2 \theta_h/\alpha^2}}{\pi \alpha^2 \cos^4 \theta_h} d\theta_h \\ &= 1 - e^{-\tan^2 \theta'/\alpha^2}. \end{aligned}$$

To find the sampling technique, we need to solve

$$\xi = 1 - e^{-\tan^2 \theta'/\alpha^2}$$

for θ' in terms of ξ . In this case, $\tan^2 \theta'$ suffices to find the microfacet orientation and is more efficient to compute, so we will compute

$$\tan^2 \theta' = -\alpha^2 \log \xi.$$

The sampling code follows directly, though we must take care of the case where $u[0]$ is zero, which causes `std::log()` to return negative infinity.

(Compute $\tan^2 \theta$ and ϕ for Beckmann distribution sample) ≡

808

```
Float tan2Theta, phi;
if (alphax == alphay) {
    Float logSample = std::log(u[0]);
    if (std::isinf(logSample)) logSample = 0;
    tan2Theta = -alphax * alphax * logSample;
    phi = u[1] * 2 * Pi;
} else {
    (Compute tan2Theta and phi for anisotropic Beckmann distribution)
}
```

The algorithm to sample $\tan^2 \theta$ and ϕ for anisotropic Beckmann–Spizzichino distributions can be derived following a similar process, though we won’t include the derivation or implementation in the text here.

Given $\tan^2 \theta$, we can compute $\cos \theta$ using the identities $\tan^2 \theta = \sin^2 \theta / \cos^2 \theta$ and $\sin^2 \theta + \cos^2 \theta = 1$. $\sin \theta$ follows, and we have enough information to compute the microfacet orientation using the spherical coordinates formula. Because pbrt transforms the normal to $(0, 0, 1)$ in the reflection coordinate system, we can almost use the computed direction from spherical coordinates directly. The last detail to handle is that if ω_o is in the opposite hemisphere than the normal, then the half-angle vector needs to be flipped to be in that hemisphere as well.

```
BeckmannDistribution::alphax
539
BeckmannDistribution::alphay
539
Float 1062
Pi 1063
SameHemisphere() 807
SphericalDirection() 346
Vector3f 60
```

(Map sampled Beckmann angles to normal direction wh) ≡

808

```
Float cosTheta = 1 / std::sqrt(1 + tan2Theta);
Float sinTheta = std::sqrt(std::max((Float)0, 1 - cosTheta * cosTheta));
Vector3f wh = SphericalDirection(sinTheta, cosTheta, phi);
if (!SameHemisphere(wo, wh)) wh = -wh;
```

While sampling a microfacet orientation from the full microfacet distribution gives correct results, the efficiency of this approach is limited by the fact that only one term, $D(\omega_h)$, of the full microfacet BSDF (defined in Equation (8.19)) is accounted for. A better approach can be found using the observation that the distribution of microfacets that

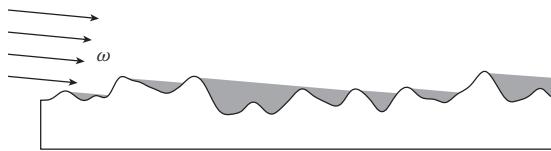


Figure 14.1: The distribution of visible microfacets from a direction oblique to the underlying geometric normal is quite different from the distribution $D(\omega_h)$. First, some microfacet orientations are backfacing and will never be seen. Others are shadowed by other microfacets. Finally, the projected area of a visible microfacet increases as its orientation approaches the viewing direction. These factors are accounted for in $D_\omega(\omega_h)$, the distribution of visible microfacets in the direction ω .

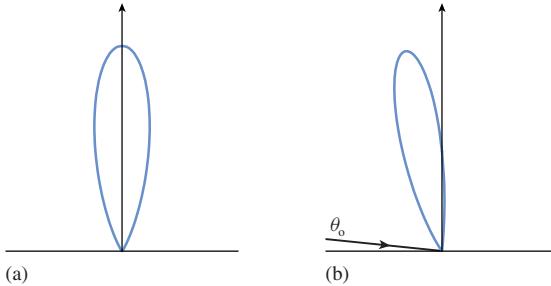


Figure 14.2: Full Beckmann-Spizzichino microfacet distribution $D(\omega_h)$ for $\alpha = 0.3$ and visible microfacet distribution $D_{\omega_o}(\omega_h)$ for $\cos \theta_o = 0.1$. With this relatively oblique viewing direction, the two distributions are quite different and sampling from $D_{\omega_o}(\omega_h)$ is a much more effective approach.

are visible from a given direction isn't the same as the full distribution of microfacets; Figure 14.1 shows the geometric setting and describes why the distributions differ.

Equation (8.13) in Section 8.4.2 defined a relationship between the visible area of microfacets from a given direction and microfacet orientation. This equation can be rearranged to get the distribution of visible normals in a direction ω :

$$D_\omega(\omega_h) = \frac{D(\omega_h) G_1(\omega, \omega_h) \max(0, \omega \cdot \omega_h)}{\cos \theta}. \quad [14.2]$$

Here, the G_1 term accounts for microfacet self-shadowing, and the $\max(0, (\omega \cdot \omega_h)) / \cos \theta$ term accounts for both backfacing microfacets and the interaction of microfacet orientation and projected area as a function of viewing direction that was shown in Figure 14.1.

Figure 14.2 compares the overall distribution of microfacets with the Beckmann-Spizzichino model with the visible distribution from a fairly oblique viewing direction. Note that many orientations are no longer visible at all (as they are backfacing) and that the microfacet orientations that are in the vicinity of the outgoing direction ω_o have a higher probability of being visible than they do in the overall distribution $D(\omega_h)$.

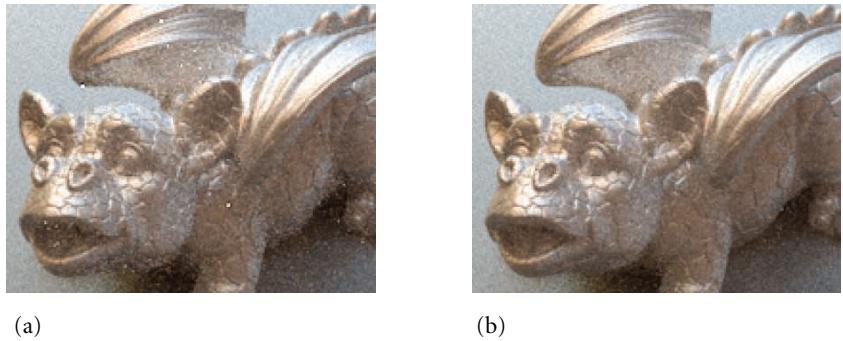


Figure 14.3: Comparison of Microfacet Sampling Techniques. With the same number of samples taken per pixel there is much higher variance when (a) sampling the full microfacet distribution $D(\omega_h)$ than when (b) sampling the visible microfacet distribution $D_{\omega_0}(\omega_h)$.

It turns out that samples can be taken directly from the distribution defined by Equation (14.2); because this distribution better matches the full Torrance–Sparrow model (Equation (8.19)) than $D(\omega_h)$ alone, there is much less variance in resulting images (see Figure 14.3). We won’t go into the extensive details of how to directly sample this distribution or include the code in the book; see the “Further Reading” section and the source code, respectively, for more information. (The `TrowbridgeReitzDistribution::Sample_wh()` method similarly samples from either the full distribution of microfacet normals or from the visible distribution; see the source code for details.)

The implementation of the `MicrofacetDistribution::Pdf()` method now follows directly; it’s just a matter of returning the density from the selected sampling distribution.

```
{MicrofacetDistribution Method Definitions} +≡
    Float MicrofacetDistribution::Pdf(const Vector3f &wo,
        const Vector3f &wh) const {
        if (sampleVisibleArea)
            return D(wh) * G1(wo) * AbsDot(wo, wh) / AbsCosTheta(wo);
        else
            return D(wh) * AbsCosTheta(wh);
    }
```

Given the ability to sample from distributions of microfacet orientations, the `MicrofacetReflection::Sample_f()` method can now be implemented.

```
{BxDF Method Definitions} +≡
    Spectrum MicrofacetReflection::Sample_f(const Vector3f &wo, Vector3f *wi,
        const Point2f &u, Float *pdf, BxDFType *sampledType) const {
        <Sample microfacet orientation ω_h and reflected direction ω_i 812>
        <Compute PDF of wi for microfacet reflection 813>
        return f(wo, *wi);
    }
```

The implementation first uses `Sample_wh()` to find a microfacet orientation and reflects the outgoing direction about the microfacet’s normal. If the reflected direction is in the

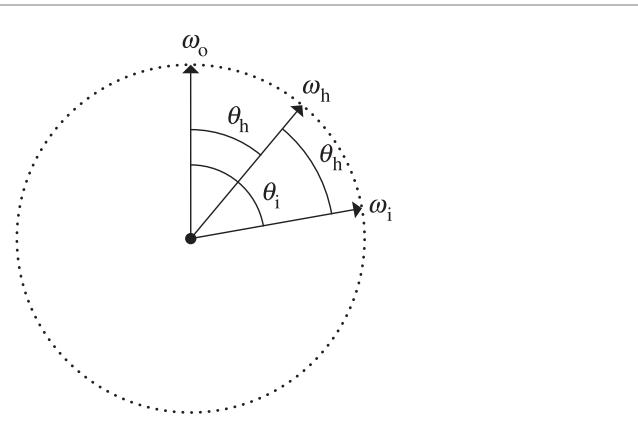


Figure 14.4: The adjustment for change of variable from sampling from the half-angle distribution to sampling from the incident direction distribution can be derived with an observation about the relative angles involved: $\theta_i = 2\theta_h$. (Note the difference in notation: for example, here we use θ_h as the angle between ω_o and ω_h ; elsewhere in the book, θ_h denotes the angle between ω_h and the surface normal.)

opposite hemisphere from ω_o , then its direction is under the surface and no light is reflected.²

(*Sample microfacet orientation ω_h and reflected direction ω_i*) ≡

811, 814

```
Vector3f wh = distribution->Sample_wh(wo, u);
*wi = Reflect(wo, wh);
if (!SameHemisphere(wo, *wi)) return Spectrum(0.f);
```

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

A simple geometric construction gives the relationship between the two distributions. Consider the spherical coordinate system oriented about ω_o (Figure 14.4). Measured with respect to ω_h , the differential solid angles $d\omega_i$ and $d\omega_h$ are $\sin \theta_i d\theta_i d\phi_i$ and $\sin \theta_h d\theta_h d\phi_h$, respectively; thus,

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

```
MicrofacetDistribution::
    Sample_wh() 807
MicrofacetReflection::
    distribution 547
    Reflect() 526
    SameHemisphere() 807
    Spectrum 315
    Vector3f 60
```

² More generally, these rays should actually intersect other microfacets and eventually be reflected, but the Torrance–Sparrow model doesn't account for microfacet interreflection.

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

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

Therefore, the PDF after transformation is

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

(Compute PDF of ω_i for microfacet reflection) 811
 $*pdf = distribution->Pdf(wo, wh) / (4 * Dot(wo, wh));$

The same computation is implemented in the `MicrofacetReflection::Pdf()` method.

```
BxDFType 513
CosTheta() 510
Float 1062
MicrofacetDistribution::Pdf()
  811
MicrofacetDistribution::
  Sample_wh()
  807
MicrofacetReflection 547
MicrofacetReflection::
  distribution
  547
MicrofacetTransmission 548
MicrofacetTransmission::
  distribution
  548
MicrofacetTransmission::etaA
  548
MicrofacetTransmission::etaB
  548
MicrofacetTransmission::f()
  548
MicrofacetTransmission::Pdf()
  814
Normal3f 71
Point2f 68
Refract() 531
SameHemisphere() 807
Spectrum 315
Vector3f::Normalize() 66
Vector3f 60
```

The approach for transmission is analogous: given a sampled ω_h microfacet orientation, the outgoing direction is refracted about that normal direction to get the sampled incident direction. In the case of total internal reflection, `Refract()` returns false, and a black SPD is returned.

(BxDF Method Definitions) +≡

```
Spectrum MicrofacetTransmission::Sample_f(const Vector3f &wo,
  Vector3f *wi, const Point2f &u, Float *pdf,
  BxDFType *sampledType) const {
  Vector3f wh = distribution->Sample_wh(wo, u);
  Float eta = CosTheta(wo) > 0 ? (etaA / etaB) : (etaB / etaA);
  if (!Refract(wo, (Normal3f)wh, eta, wi))
    return 0;
  *pdf = Pdf(wo, *wi);
  return f(wo, *wi);
}
```

The PDF for the transmitted direction also requires an adjustment for the change of variables. This value is stored in `dwh_dwi`. We won't derive this term here; the "Further Reading" section at the end of the chapter has more information.

```
(BxDF Method Definitions) +≡
Float MicrofacetTransmission::Pdf(const Vector3f &wo,
    const Vector3f &wi) const {
    if (SameHemisphere(wo, wi)) return 0;
    (Compute  $\omega_h$  from  $\omega_o$  and  $\omega_i$  for microfacet transmission 814)
    (Compute change of variables  $d\omega_h \cdot d\omega_i$  for microfacet transmission)
    return distribution->Pdf(wo, wh) * dwh_dwi;
}
```

In the transmissive case, the meaning of the half-angle vector ω_h is generalized to denote the normal of the microfacet that is responsible for refracting light from ω_i to ω_o . This vector can be derived following the setting in Figure 8.11.

(Compute ω_h from ω_o and ω_i for microfacet transmission) ≡ 814

```
Float eta = CosTheta(wo) > 0 ? (etaB / etaA) : (etaA / etaB);
Vector3f wh = Normalize(wo + wi * eta);
```

14.1.2 FresnelBlend

The FresnelBlend class is a mixture of a diffuse and glossy term. A straightforward approach to sampling this BRDF is to sample from both a cosine-weighted distribution as well as the microfacet distribution. The implementation here chooses between the two with equal probability based on whether ξ_1 is less than or greater than 0.5. In both cases, it remaps ξ_1 to cover the range [0, 1) after using it to make this decision. (Otherwise, values of ξ_1 used for the cosine-weighted sampling would always be less than 0.5, for example.) Using the sample ξ_1 for two purposes in this manner slightly reduces the quality of the stratification of the (ξ_1, ξ_2) values that are actually used for sampling directions.³

```
(BxDF Method Definitions) +≡
Spectrum FresnelBlend::Sample_f(const Vector3f &wo, Vector3f *wi,
    const Point2f &uOrig, Float *pdf, BxDFType *sampledType) const {
    Point2f u = uOrig;
    if (u[0] < .5) {
        u[0] = 2 * u[0];
        (Cosine-sample the hemisphere, flipping the direction if necessary 806)
    } else {
        u[0] = 2 * (u[0] - .5f);
        (Sample microfacet orientation  $\omega_h$  and reflected direction  $\omega_i$  812)
    }
    *pdf = Pdf(wo, *wi);
    return f(wo, *wi);
}
```

BSDF::Sample_f() 832
 BxDFType 513
 CosTheta() 510
 Float 1062
 FresnelBlend 550
 FresnelBlend::f() 551
 FresnelBlend::Pdf() 815
 MicrofacetDistribution::Pdf() 811
 MicrofacetTransmission 548
 MicrofacetTransmission::distribution 548
 MicrofacetTransmission::etaA 548
 MicrofacetTransmission::etaB 548
 Point2f 68
 SameHemisphere() 807
 Spectrum 315
 Vector3f::Normalize() 66
 Vector3f 60

³ Alternatively, we could have modified the interface of Sample_f() to supply an additional sample for such discrete choices, including those decisions made in BSDF::Sample_f(). However, consuming additional dimensions of the sample vector can also have an adverse effect on the quality of subsequent samples that end up coming from higher dimensions as a consequence—which approach is preferable in practice is related to the type of sampler being used.

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

(BxDF Method Definitions) +≡

```
Float FresnelBlend::Pdf(const Vector3f &wo, const Vector3f &wi) const {
    if (!SameHemisphere(wo, wi)) return 0;
    Vector3f wh = Normalize(wo + wi);
    Float pdf_wh = distribution->Pdf(wo, wh);
    return .5f * (AbsCosTheta(wi) * InvPi +
                  pdf_wh / (4 * Dot(wo, wh)));
}
```

14.1.3 SPECULAR REFLECTION AND TRANSMISSION

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

Recall that the Dirac delta distribution is defined such that

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

and

$$\int_{-\infty}^{\infty} \delta(x) dx = 1.$$

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

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

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

$$\frac{1}{N} \sum_i^N \frac{f_r(p, \omega_o, \omega_i) L_i(p, \omega_i) |\cos \theta_i|}{p(\omega_i)} = \frac{1}{N} \sum_i^N \frac{\rho_{hd}(\omega_o) \frac{\delta(\omega - \omega_i)}{|\cos \theta_i|} L_i(p, \omega_i) |\cos \theta_i|}{p(\omega_i)},$$

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

AbsCosTheta() 510
 BxDF 513
 Float 1062
 FresnelBlend 550
 FresnelBlend::distribution 550
 InvPi 1063
 MicrofacetDistribution::Pdf() 811
 SameHemisphere() 807
 SpecularReflection 524
 SpecularTransmission 528
 Vector3f::Normalize() 66
 Vector3f 60

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

$$\rho_{hd}(\omega_o)L_i(p, \omega),$$

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

Therefore, the implementations here return a constant value of 1 for the PDF for specular reflection and transmission when sampled using `Sample_f()`, with the convention that for specular BxDFs there is an implied delta distribution in the PDF value that is expected to cancel out with the implied delta distribution in the value of the BSDF when the estimator is evaluated. The respective `Pdf()` methods therefore return 0 for all directions, since there is zero probability that another sampling method will randomly find the direction from a delta distribution.

(SpecularReflection Public Methods) +≡

524

```
Float Pdf(const Vector3f &wo, const Vector3f &wi) const {
    return 0;
}
```

(SpecularTransmission Public Methods) +≡

528

```
Float Pdf(const Vector3f &wo, const Vector3f &wi) const {
    return 0;
}
```

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

The `FresnelSpecular` class encapsulates both specular reflection and transmission, with the relative contributions modulated by a dielectric Fresnel term. By combining these two together, it's able to use the value of the Fresnel term for the outgoing direction ω_o to determine which component to sample—for example, for glancing angles where reflection is high, it's much more likely to return a reflected direction than a transmitted direction. This approach improves Monte Carlo efficiency when rendering scenes with these sorts of surfaces, since the rays that are sampled will tend to have larger contributions to the final result.

(BxDF Method Definitions) +≡

```
Spectrum FresnelSpecular::Sample_f(const Vector3f &wo,
    Vector3f *wi, const Point2f &u, Float *pdf,
    BxDFType *sampledType) const {
    Float F = FrDielectric(CosTheta(wo), etaA, etaB);
    if (u[0] < F) {
        <Compute specular reflection for FresnelSpecular 817>
    } else {
        <Compute specular transmission for FresnelSpecular 817>
    }
}
```

BxDF 513
 BxDFType 513
 CosTheta() 510
 Float 1062
 FrDielectric() 519
 FresnelSpecular 531
 Point2f 68
 Spectrum 315
 Vector3f 60



(a)

(b)

Figure 14.5: The Effect of Accurately Sampling the FourierBSDF. Reflection from both objects is modeled using the FourierBSDF, rendered using 32 samples per pixel. (a) Uniform hemisphere sampling to compute reflection. (b) The exact sampling scheme implemented in FourierBSDF::Sample_f(). Variance is much lower and overall rendering time increased by only 20%.

Specular reflection is chosen with probability equal to F ; given that choice, the computations performed are the same as those in SpecularReflection.

```
(Compute specular reflection for FresnelSpecular) ≡
  (Compute perfect specular reflection direction 526)
  if (sampledType)
    *sampledType = BxDFType(BSDF_SPECULAR | BSDF_REFLECTION);
  *pdf = F;
  return F * R / AbsCosTheta(*wi);
```

816

Otherwise, with probability $1-F$, specular transmission is selected.

```
(Compute specular transmission for FresnelSpecular) ≡
  (Figure out which  $\eta$  is incident and which is transmitted 529)
  (Compute ray direction for specular transmission 529)
  Spectrum ft = T * (1 - F);
  (Account for non-symmetry with transmission to different medium 961)
  if (sampledType)
    *sampledType = BxDFType(BSDF_SPECULAR | BSDF_TRANSMISSION);
    *pdf = 1 - F;
    return ft / AbsCosTheta(*wi);
```

816

AbsCosTheta() 510
 BSDF_REFLECTION 513
 BSDF_SPECULAR 513
 BSDF_TRANSMISSION 513
 BxDFType 513
 FourierBSDF 555
 FourierBSDF::Sample_f() 819
 FresnelSpecular::R 532
 FresnelSpecular::T 532
 Spectrum 315
 SpecularReflection 524

★ 14.1.4 FOURIER BSDF

In addition to being able to compactly and accurately represent a variety of measured and synthetic BSDFs, the representation used by the FourierBSDF (Section 8.6) also admits a fairly efficient exact importance sampling scheme. Figure 14.5 compares the result of using this sampling approach to using uniform hemispherical sampling for the FourierBSDF.

Recall the BSDF representation from Equation (8.22), which was

$$f(\mu_i, \mu_o, \phi_i - \phi_o) |\mu_i| = \sum_{k=0}^{m-1} a_k(\mu_i, \mu_o) \cos(k(\phi_i - \phi_o)),$$

where the function a_k was discretized over the incident and outgoing zenith angle cosines $(\mu_i, \mu_o) \in \{\mu_0, \dots, \mu_{n-1}\}^2$ with endpoints $\mu_0 = -1$ and $\mu_{n-1} = 1$. An even Fourier expansion with real coefficients was used to model the dependence on the azimuth angle difference parameter $\phi = \phi_i - \phi_o$.

The task now is to first sample μ_i given μ_o and then sample the angle ϕ_i relative to ϕ_o . A helpful property of the order 0 Fourier coefficients greatly simplifies both of these steps. The even Fourier basis functions form an orthogonal basis of the vector space of square-integrable even functions—this means that the basis coefficients of any function g satisfying these criteria can be found using a inner product between g and the individual basis functions analogous to orthogonal projections of vectors on Euclidean vector spaces. Here, we are dealing with continuous functions on $[0, \pi]$, where a suitable inner product can be defined as

$$\langle g, h \rangle = \frac{1}{\pi} \int_0^\pi g(\phi) h(\phi) d\phi.$$

The Fourier basis function associated with order 0 is simply the unit constant; hence the coefficients in a_0 relate to the cosine-weighted BSDF as

$$a_0(\mu_i, \mu_o) = \frac{1}{\pi} \int_0^\pi f(\mu_i, \mu_o, \phi) |\mu_i| d\phi.$$

This quantity turns out to be very helpful in constructing a method for importance sampling the BSDF: disregarding normalization factors, this average over ϕ can be interpreted as the marginal distribution of the cosine-weighted BSDF with respect to pairs of μ angles (Section 13.6 discussed marginal density functions).

It will be useful to be able to efficiently access these order 0 coefficients without the in-directions that would normally be necessary given the layout of `FourierBSDFTable::a`. We therefore keep an additional copy of these values in an array of size $nMu \times nMu$ in `FourierBSDFTable::a0`. This array is initialized by copying the corresponding entries from `FourierBSDFTable::a` at scene loading time in the `FourierBSDFTable::Read()` method.

(FourierBSDFTable Public Data) +≡

554

`Float *a0;`

With a marginal distribution at hand, we are now able to split the sampling operation into two lower dimensional steps: first, we use the a_0 coefficients to sample μ_i given μ_o . Second, with (μ_i, μ_o) known, we can interpolate the Fourier coefficients that specify the BSDF's dependence on the remaining azimuth difference angle parameter ϕ and sample from their distribution. These operations are all implemented as smooth mappings that preserve the properties of structured point sets, such as Sobol' or Halton sequences. Given these angles, the last step is to compute the corresponding direction and value of the BSDF.

`Float` 1062

`FourierBSDFTable::a` 555

`FourierBSDFTable::a0` 818

`FourierBSDFTable::Read()` 554

```

⟨BxDF Method Definitions⟩ +≡
    Spectrum FourierBSDF::Sample_f(const Vector3f &wo, Vector3f *wi,
        const Point2f &u, Float *pdf, BxDFType *sampledType) const {
        ⟨Sample zenith angle component for FourierBSDF 819⟩
        ⟨Compute Fourier coefficients  $a_k$  for  $(\mu_i, \mu_o)$  557⟩
        ⟨Importance sample the luminance Fourier expansion 820⟩
        ⟨Compute the scattered direction for FourierBSDF 820⟩
        ⟨Evaluate remaining Fourier expansions for angle  $\phi$ ⟩
    }
}

```

Sampling the zenith angle is implemented using `SampleCatmullRom2D()`, which will be defined in a few pages. This helper function operates in stages: after first mapping a uniform variate to one of the spline segments, it then samples a specific position within the segment. To select a segment, the function requires an array of precomputed CDFs

$$I_{i,o} = \int_{-1}^{\mu_i} a_0(\mu', \mu_o) d\mu', \quad [14.3]$$

where $0 \leq i, o < n$. Each column of this matrix stores a discrete CDF over μ_i for a different (fixed) value of μ_o . The above integral is computed directly from the spline interpolant and can thus be used to efficiently select spline segments proportional to their definite integral.

⟨FourierBSDFTable Public Data⟩ +≡

554

```
Float *cdf;
```

In the case of the FourierBSDF, this `cdf` array is already part of the input file, and we need not be concerned with its generation.

⟨Sample zenith angle component for FourierBSDF⟩ ≡

819

```

Float mu0 = CosTheta(wo);
Float pdfMu;
Float muI = SampleCatmullRom2D(bsdfTable.nMu, bsdfTable.nMu,
    bsdfTable.mu, bsdfTable.mu, bsdfTable.a0, bsdfTable.cdf, mu0,
    u[1], nullptr, &pdfMu);
}

```

After `SampleCatmullRom2D()` returns, `muI` records the sampled incident zenith angle cosine, and `pdfMu` contains the associated probability density in the same domain.

```

BxDFType 513
CosTheta() 510
Float 1062
FourierBSDF 555
FourierBSDF::bsdfTable 556
FourierBSDF::f() 556
FourierBSDFTable::a0 818
FourierBSDFTable::cdf 819
FourierBSDFTable::mu 554
FourierBSDFTable::nMu 554
Point2f 68
SampleCatmullRom2D() 824
SampleFourier() 828
Spectrum 315
Vector3f 60

```

We can now interpolate the nearby Fourier coefficients, reusing the fragment *⟨Compute Fourier coefficients a_k for (μ_i, μ_o) ⟩* from `FourierBSDF::f()` in Section 8.6. Given the coefficients a_k , sampling of the azimuth difference angle is also implemented as a separate function `SampleFourier()`, also to be defined in a few pages. This function returns the sampled difference angle ϕ , the value Y of the luminance Fourier expansion evaluated at ϕ , and the sample probability $pdfPhi$ per unit radian. The final sample probability per unit solid angle is the product of the azimuthal and zenith angle cosine PDFs. (As with values computed via Fourier series in Section 8.6, negative values must be clamped to 0.)

(Importance sample the luminance Fourier expansion) ≡ 819

```
Float phi, pdfPhi;
Float Y = SampleFourier(ak, bsdfTable.recip, mMax, u[0], &pdfPhi, &phi);
*pdf = std::max((Float)0, pdfPhi * pdfMu);
```

SampleFourier() takes an additional input array recip containing precomputed integer reciprocals $1/i$ for all mMax Fourier orders. These reciprocals are frequently accessed within the function—precomputing them is an optimization to avoid costly arithmetic to generate them over and over again, causing pipeline stalls due to the high latency of division operations on current processor architectures. This reciprocal array is initialized in FourierBSDFTable::Read().

(FourierBSDFTable Public Data) +≡ 554

```
Float *recip;
```

We now have the angles $\mu_i = \cos \theta_i$ and ϕ . The sampled incident direction's xy coordinates are given by rotating the xy components of ω_o by an angle ϕ about the surface normal, and its z component is given by μ_i (using spherical coordinates).

There are two details to note in the computation of the direction ω_i . First, the xy components are scaled by a factor $\sin \theta_i / \sin \theta_o$, which ensures that the resulting vector is normalized. Second, the computed direction is negated before being assigned to *wi; this follows the coordinate system convention for the FourierBSDF that was described in Section 8.6.

(Compute the scattered direction for FourierBSDF) ≡ 819

```
Float sin2ThetaI = std::max((Float)0, 1 - muI * muI);
Float norm = std::sqrt(sin2ThetaI / Sin2Theta(wo));
Float sinPhi = std::sin(phi), cosPhi = std::cos(phi);
*wi = -Vector3f(norm * (cosPhi * wo.x - sinPhi * wo.y),
                 norm * (sinPhi * wo.x + cosPhi * wo.y), muI);
```

The fragment *(Evaluate remaining Fourier expansions for angle ϕ)* is identical to *(Evaluate Fourier expansion for angle ϕ)* defined in Section 8.6 except that doesn't evaluate the luminance channel, which was already done by SampleFourier() above.

The FourierBSDF::Pdf() method returns the solid angle density for the preceding sampling method. Since this method produces samples that are exactly distributed according to the product $f(\mu_i, \mu_o, \phi) |\mu_i|$, we could simply copy the implementation of FourierBSDF::f() except for the division that cancels $|\mu_i|$. However, doing so would underestimate the probability when the BSDF doesn't reflect all of the incident illumination.

To correct for this, we scale the unnormalized $f(\mu_i, \mu_o, \phi) |\mu_i|$ by a suitable normalization factor ρ^{-1} to ensure that the product integrates to 1:

$$\int_0^{2\pi} \int_0^{\pi} \frac{1}{\rho} f(\cos \theta_i, \cos \theta_o, \phi) |\cos \theta_i| \sin \theta_i d\theta'_i d\phi = 1.$$

Note that the outgoing zenith angle cosine $\cos \theta_o = \mu_o$ was left unspecified in the above equation. In general, the normalization factor ρ is not constant and, instead, it depends on the current value of μ_o . $\rho(\mu_o)$ has a simple interpretation: it is the hemispherical-directional reflectance of a surface that is illuminated from the zenith angle $\cos^{-1} \mu_o$.

Float 1062

FourierBSDF 555

FourierBSDF::bsdfTable 556

FourierBSDF::f() 556

FourierBSDFTable::Read() 554

FourierBSDFTable::recip 820

SampleFourier() 828

Vector3f 60

Suppose briefly that μ_o happens to be part of the discretized set of zenith angle cosines μ_0, \dots, μ_{n-1} stored in the array `FourierBSDFTable::mu`. Then

$$\begin{aligned}\rho(\mu_o) &= \int_0^{2\pi} \int_0^\pi f(\cos \theta_i, \cos \theta_o, \phi) |\cos \theta_i| \sin \theta_i d\theta_i d\phi \\ &= \int_0^{2\pi} \int_{-1}^1 f(\mu_i, \mu_o, \phi) |\mu_i| d\mu_i d\phi \\ &= 2\pi \int_{-1}^1 \left[\frac{1}{\pi} \int_0^\pi f(\mu_i, \mu_o, \phi) |\mu_i| d\phi \right] d\mu_i \\ &= 2\pi \int_{-1}^1 a_0(\mu_i, \mu_o) d\mu_i \\ &= 2\pi I_{n-1, o},\end{aligned}\tag{14.4}$$

where I was defined in Equation (14.3). In other words, the needed normalization factor is readily available in the `FourierBSDFTable::cdf` array. For intermediate values of μ_o , we can simply interpolate the neighboring four entries of $I_{n-1, i}$ using the usual spline interpolation scheme—the linearity of this interpolation coupled with the linearity of the analytic integrals in (14.4) ensures a result that is consistent with `FourierBSDF::f()`.

(BxDF Method Definitions) \equiv

```
Float FourierBSDF::Pdf(const Vector3f &wo, const Vector3f &wi) const {
    <Find the zenith angle cosines and azimuth difference angle 556>
    <Compute luminance Fourier coefficients a_k for (<math>\mu_i, \mu_o</math>)>
    <Evaluate probability of sampling wi 821>
}
```

We won't include the second fragment here—it is almost identical to *(Compute Fourier coefficients a_k for (μ_i, μ_o))*, the only difference being that it only interpolates the luminance coefficients (samples are generated proportional to luminance; hence the other two channels are not relevant here).

The last fragment interpolates the directional albedo from Equation (14.4) and uses it to correct the result of `Fourier()` for absorption.

(Evaluate probability of sampling wi) \equiv

```
821
Float rho = 0;
for (int o = 0; o < 4; ++o) {
    if (weights0[o] == 0)
        continue;
    rho += weights0[o] * bsdfTable.cdf[(offset0 + o) * bsdfTable.nMu +
                                         bsdfTable.nMu - 1] * (2 * Pi);
}
Float Y = Fourier(ak, mMax, cosPhi);
return (rho > 0 && Y > 0) ? (Y / rho) : 0;
```

Sampling 1D Spline Interpolants

Before defining the `SampleCatmullRom2D()` function used in the previously discussed `FourierBSDF::Sample_f()` method, we'll first focus on a simpler 1D case: suppose that

a function f was evaluated at n positions x_0, \dots, x_{n-1} , resulting in a piecewise cubic Catmull-Rom spline interpolant \hat{f} with $n - 1$ spline segments $\hat{f}_i(x)$ ($i = 0, \dots, n - 2$). Given a precomputed discrete CDF over these spline segments defined as

$$F_i = \begin{cases} 0, & \text{if } i = 0, \\ \sum_{k=0}^{i-1} \int_{x_k}^{x_{k+1}} \frac{1}{c} \hat{f}_k(x') dx', & \text{if } i > 0, \end{cases} \quad [14.5]$$

where c is the normalization term,

$$c = \int_{x_0}^{x_{n-1}} \hat{f}(x) dx,$$

a straightforward way of importance sampling \hat{f} in two steps using the inversion method entails first finding the interval i such that

$$F_i \leq \xi_1 < F_{i+1},$$

where ξ_1 is a random variate on the interval $[0, 1]$, and then sampling an x' value in the i th interval. Since the values F_i are monotonically increasing, the interval can be found using an efficient binary search.

In the following, we won't actually normalize the F_i values, effectively setting $c = 1$. We can equivalently sample i by first multiplying the random variable ξ by the last F_i entry, F_{n-1} , which is the total integral over all spline segments and is thus equal to c . Thus, the binary search looks for

$$F_i \leq \xi_1 F_{n-1} \leq F_{i+1}, \quad [14.6]$$

Having selected a segment i , we can offset and re-scale ξ_1 to obtain a second uniform variate in $[0, 1)$:

$$\xi_2 = \frac{\xi_1 F_{n-1} - F_i}{F_{i+1} - F_i}.$$

We then use ξ_2 to sample a position x within the interval $[x_i, x_{i+1}]$ using that segment's integral,

$$\hat{F}_i(x) = \int_{x_i}^x \hat{f}_i(x') dx', \quad [14.7]$$

where again we won't compute a properly normalized CDF but will instead multiply ξ_2 by $\hat{F}_i(x_{i+1})$ rather than normalizing \hat{F}_i . We must then compute

$$\begin{aligned} x &= \hat{F}_i^{-1} \left(\hat{F}_i(x_{i+1}) \xi_2 \right) = \hat{F}_i^{-1} \left((F_{i+1} - F_i) \frac{\xi_1 F_{n-1} - F_i}{F_{i+1} - F_i} \right) \\ &= \hat{F}_i^{-1} (\xi_1 F_{n-1} - F_i). \end{aligned} \quad [14.8]$$

This approach is implemented in `SampleCatmullRom()`, which takes the following inputs: the number of samples n ; x contains the locations x_0, \dots, x_{n-1} where the original function f was evaluated; f stores the value of the function at each point x_i ; u is used to pass the uniform variate ξ ; and integrated F_i values must be provided via the F parameter—

`SampleCatmullRom()` 823

these values can be precomputed with `IntegrateCatmullRom()` when necessary. `fval` and `pdf` are used to return the function value and associated PDF value.

```
(Spline Interpolation Definitions) +≡
    Float SampleCatmullRom(int n, const Float *x, const Float *f,
                           const Float *F, Float u, Float *fval, Float *pdf) {
        (Map u to a spline interval by inverting F 823)
        (Look up  $x_i$  and function values of spline segment i 823)
        (Approximate derivatives using finite differences 823)
        (Re-scale u for continuous spline sampling step 824)
        (Invert definite integral over spline segment and return solution 825)
    }
```

The function begins by scaling the uniform variate u by the last entry of F following Equation (14.6). Following this, u is mapped to a spline interval via the `FindInterval()` helper function, which returns the last interval satisfying $F[i] \leq u$ while clamping to the array bounds in case of rounding errors.

```
(Map u to a spline interval by inverting F) ≡ 823
    u *= F[n - 1];
    int i = FindInterval(n, [&](int i) { return F[i] <= u; });


```

The next fragment fetches the associated function values and node positions from f and x ; the variable `width` contains the segment length.

```
(Look up  $x_i$  and function values of spline segment i) ≡ 823, 937
    Float x0 = x[i], x1 = x[i + 1];
    Float f0 = f[i], f1 = f[i + 1];
    Float width = x1 - x0;
```

Recall that Catmull-Rom splines require an approximation of the first derivative of the function (Section 8.6.1) at the segment endpoints. Depending on i , this derivative is computed using forward, backward, or central finite difference approximations.

```
(Approximate derivatives using finite differences) ≡ 823, 937
    Float d0, d1;
    if (i > 0)      d0 = width * (f1 - f[i - 1]) / (x1 - x[i - 1]);
    else            d0 = f1 - f0;
    if (i + 2 < n) d1 = width * (f[i + 2] - f0) / (x[i + 2] - x0);
    else            d1 = f1 - f0;
```

The remainder of the function then has to find the inverse of the continuous cumulative distribution function from Equation (14.8):

```
FindInterval() 1065
Float 1062
IntegrateCatmullRom() 937
```

$$F_i^{-1}(\xi_1 F_{n-1} - F_i).$$

Since the scaling by F_{n-1} was already applied in the first fragment, we need only subtract F_i .

The actual inversion is done in *(Invert definite integral over spline segment and return solution)*, whose discussion we postpone for the following discussion of the 2D cases. The internals of this inversion operate on a scaled and shifted spline segment defined on the interval $[0, 1]$, which requires an additional scaling by the associated change of variable factor equal to the reciprocal of width.

(Re-scale u for continuous spline sampling step) \equiv 823
 $u = (u - F[i]) / \text{width};$

Sampling 2D Spline Interpolants

The main use cases of spline interpolants in pbrt actually importance sample 2D functions $f(\alpha, x)$, where α is considered a fixed parameter for the purpose of sampling (e.g., the albedo of the underlying material or the outgoing zenith angle cosine μ_o in the case of the FourierBSDF). This case is handled by `SampleCatmullRom2D()`.

(Spline Interpolation Definitions) $+ \equiv$
`Float SampleCatmullRom2D(int size1, int size2, const Float *nodes1,
 const Float *nodes2, const Float *values, const Float *cdf,
 Float alpha, Float u, Float *fval, Float *pdf) {`
(Determine offset and coefficients for the alpha parameter 824)
(Define a lambda function to interpolate table entries 825)
(Map u to a spline interval by inverting the interpolated cdf)
(Look up node positions and interpolated function values)
(Re-scale u using the interpolated cdf)
(Approximate derivatives using finite differences of the interpolant)
(Invert definite integral over spline segment and return solution 825)
`}`

The parameters `size1`, `size2`, `nodes1`, and `nodes2` specify separate discretizations for each dimension. The `values` argument supplies a matrix of function values in row-major order, with rows corresponding to sets of samples that have the same position along the first dimension. The function uses the parameter `alpha` to choose a slice in the first dimension that is then importance sampled along the second dimension. The parameter `cdf` supplies a matrix of discrete CDFs, where each row was obtained by running `IntegrateCatmullRom()` on the corresponding row of `values`.

The first fragment of `SampleCatmullRom2D()` calls `CatmullRomWeights()` to select four adjacent rows of the `values` array along with interpolation weights.

(Determine offset and coefficients for the alpha parameter) \equiv 824
`int offset;
 Float weights[4];
 if (!CatmullRomWeights(size1, nodes1, alpha, &offset, weights))
 return 0;`

To proceed, we could now simply interpolate the selected rows of `values` and `cdf` and finish by calling the 1D sampling function `SampleCatmullRom()`. However, only a few entries of `values` and `cdf` are truly needed to generate a sample in practice, making

`CatmullRomWeights()` [562](#)
`Float` [1062](#)
`FourierBSDF` [555](#)
`IntegrateCatmullRom()` [937](#)
`SampleCatmullRom()` [823](#)
`SampleCatmullRom2D()` [824](#)

such an approach unnecessarily slow. Instead, we define a C++11 lambda function that interpolates entries of these arrays on demand:

```
(Define a lambda function to interpolate table entries) ≡ 824
auto interpolate = [&](const Float *array, int idx) {
    Float value = 0;
    for (int i = 0; i < 4; ++i)
        if (weights[i] != 0)
            value += array[(offset + i) * size2 + idx] * weights[i];
    return value;
};
```

The rest of the function is identical to `SampleCatmullRom()` except that every access to `values[i]` is replaced by `interpolate(values, i)` and similarly for `cdf`. For brevity, this code is omitted in the book.

We now return to the inversion of the integral in Equation (14.8), which we glossed over. Recall that \hat{F}_i was defined as an integral over the cubic spline segment \hat{f}_i , making it a quartic polynomial. It is possible but burdensome to invert this function analytically. We prefer a numerical approach that is facilitated by a useful pair of properties:

1. The function \hat{F}_i is the definite integral over the (assumed nonnegative) interpolant \hat{f}_i , and so it increases monotonically.
2. The interval $[x_i, x_{i+1}]$ selected by the function `FindInterval()` contains exactly one solution to Equation (14.8).

In this case, the interval $[x_i, x_{i+1}]$ is known as a *bracketing interval*. The existence of such an interval allows using *bisection search*, a simple iterative root-finding technique that is guaranteed to converge to the solution. In each iteration, bisection search splits the interval into two parts and discards the subinterval that does not bracket the solution—in this way, it can be interpreted as a continuous extension of binary search. The method's robustness is clearly desirable, but its relatively slow (linear) convergence can still be improved. We use *Newton-Bisection*, which is a combination of the quadratically converging but potentially unsafe⁴ Newton's method with the safety of bisection search as a fallback.

As mentioned earlier, all of the following steps assume that the spline segment under consideration is defined on the interval $[0, 1]$ with endpoint values f_0 and f_1 and derivative estimates d_0 and d_1 . We will use the variable t to denote positions in this shifted and scaled interval and the values a and b store the current interval extent. `Fhat` stores the value of $\hat{F}(t)$ and `fhat` stores $\hat{f}(t)$.

```
(Invert definite integral over spline segment and return solution) ≡ 823, 824
(Set initial guess for t by importance sampling a linear interpolant 826)
Float a = 0, b = 1, Fhat, fhat;
```

`FindInterval()` 1065

`Float` 1062

`SampleCatmullRom()` 823

⁴ Newton's method can exhibit oscillatory or divergent behavior and is only guaranteed to converge when started sufficiently close to the solution. In practice, it is usually hard to provide such a guarantee; hence we prefer the unconditionally safe combination with bisection search.

```

while (true) {
    ⟨Fall back to a bisection step when t is out of bounds 826⟩
    ⟨Evaluate target function and its derivative in Horner form 827⟩
    ⟨Stop the iteration if converged 827⟩
    ⟨Update bisection bounds using updated t 828⟩
    ⟨Perform a Newton step 828⟩
}
⟨Return the sample position and function value 828⟩

```

The number of required Newton-Bisection iterations can be reduced by starting the algorithm with a good initial guess. We use a heuristic that assumes that the spline segment is linear, i.e.,

$$\hat{f}(t) = (1-t)f(0) + tf(1).$$

Then the definite integral

$$\hat{F}(t) = \int_0^t \hat{f}(t') dt' = \frac{t}{2}(tf(1) - (t-2)f(0))$$

has the inverse

$$\hat{F}^{-1}(\xi) = \begin{cases} \frac{f(0) \pm \sqrt{f(0)^2 - 2f(0)\xi + 2f(1)\xi}}{f(0) - f(1)} & f(0) \neq f(1) \\ \frac{\xi}{f(0)} & \text{otherwise,} \end{cases}$$

of which only one of the quadratic roots is relevant (the other one yields values outside of $[0, 1]$).

(Set initial guess for t by importance sampling a linear interpolant) ≡ 825

```

Float t;
if (f0 != f1)
    t = (f0 - std::sqrt(
        std::max((Float)0, f0 * f0 + 2 * u * (f1 - f0))) / (f0 - f1));
else
    t = u / f0;

```

The first fragment in the inner loop checks if the current iterate is inside the bracketing interval $[a, b]$. Otherwise it is reset to the interval center, resulting in a standard bisection step (Figure 14.6).

(Fall back to a bisection step when t is out of bounds) ≡ 825

```

if (!(t >= a && t <= b))
    t = 0.5f * (a + b);

```

Next, F is initialized by evaluating the quartic $\hat{F}(t)$ from Equation (14.7). For Newton's method, we also require the derivative of this function, which is simply the original cubic spline—thus, f is set to the spline evaluated at t . The following expressions result after converting both functions to Horner form:

Float 1062

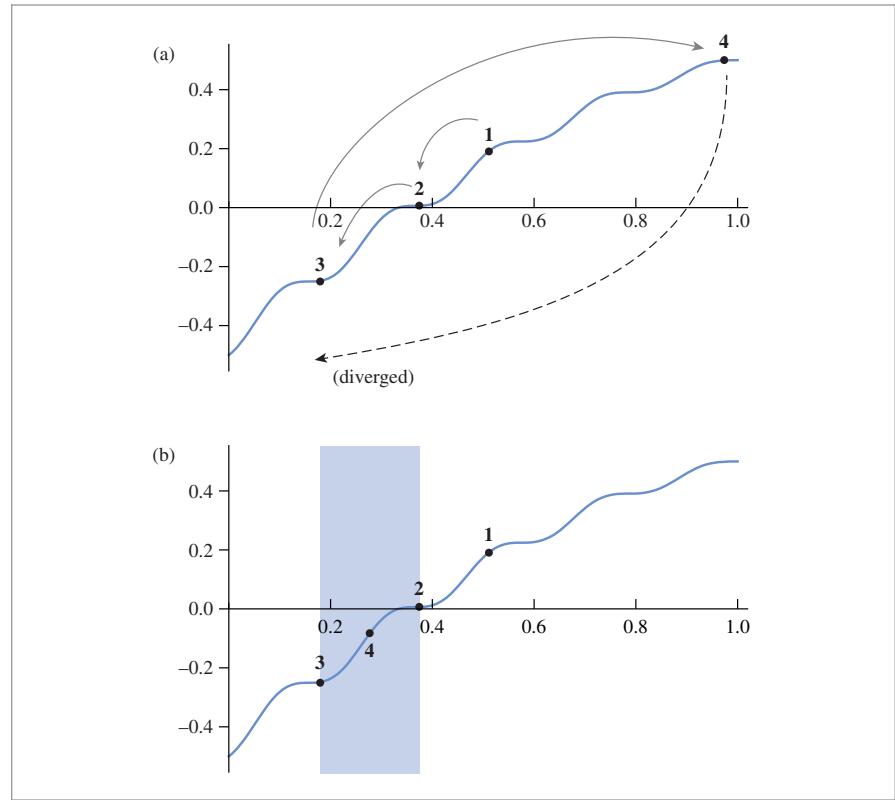


Figure 14.6: The Robustness of Newton-Bisection. (a) this function increases monotonically and contains a single root on the shown interval, but a naive application of Newton's method diverges. (b) the bisection feature of the robust root-finder enables recovery from the third Newton step, which jumps far away from the root (the bisection interval is highlighted). The method converges a few iterations later.

(Evaluate target function and its derivative in Horner form) ≡

```
Fhat = t * (f0 + t * (.5f * d0 + t * ((1.f/3.f) * (-2 * d0 - d1) +
      f1 - f0 + t * (.25f * (d0 + d1) + .5f * (f0 - f1))));

fhat = f0 + t * (d0 + t * (-2 * d0 - d1 + 3 * (f1 - f0) +
      t * (d0 + d1 + 2 * (f0 - f1))));
```

825

The iteration should stop either if $Fhat - u$ is close to 0, or if the bracketing interval has become sufficiently small.

(Stop the iteration if converged) ≡

```
if (std::abs(Fhat - u) < 1e-6f || b - a < 1e-6f)
    break;
```

825

If $\hat{F}(t) - u < 0$, then the monotonicity of \hat{F} implies that the interval $[a, t]$ cannot possibly contain the solution (and a similar statement holds for b). The next fragment uses this information to update the bracketing interval.

```
{Update bisection bounds using updated t} ≡ 825
  if (Fhat - u < 0) a = t;
  else             b = t;
```

Finally, the function and derivative values are used in a Newton step.

```
{Perform a Newton step} ≡ 825
  t -= (Fhat - u) / fhat;
```

Once converged, the last fragment maps t back to the original interval. The function optionally returns the spline value and probability density at this position.

```
{Return the sample position and function value} ≡ 825
  if (fval) *fval = fhat;
  if (pdf)  *pdf = fhat / F[n - 1];
  return x0 + width * t;
```

Sampling Fourier Expansions

Next, we'll discuss sample generation for the Fourier series, Equation (8.22) using a Newton-Bisection-type method that is very similar to what was used in `SampleCatmullRom()`. We'd like to sample from a distribution that matches

$$f(\phi) = \sum_{k=0}^{m-1} a_k \cos(k \phi)$$

for given Fourier coefficients a_k . Integrating gives a simple analytic expression:

$$F(\phi) = \int_0^\phi f(\phi') d\phi' = a_0 \phi + \sum_{k=1}^{m-1} \frac{1}{k} \sin(k \phi), \quad [14.9]$$

though note that this isn't necessarily a normalized CDF: $F(2\pi) = 2\pi a_0$, since the $\sin(k\phi)$ terms are all zero at $\phi = 2\pi$.

The function `SampleFourier()` numerically inverts $F(\phi)$ to sample azimuths using the inversion method. It takes an array `ak` of Fourier coefficients of length `m` as input. The `u` parameter is used to pass a uniform variate, and `recip` should be a pointer to an array of `m` integer reciprocals. `SampleFourier()` returns the value of the Fourier expansion at the sampled position, which is stored in `*phiPtr` along with a probability density in `*pdf`.

```
{Fourier Interpolation Definitions} +≡
  Float SampleFourier(const Float *ak, const Float *recip, int m, Float u,
                      Float *pdf, Float *phiPtr) {
    (Pick a side and declare bisection variables 829)
    while (true) {
      (Evaluate F(ϕ) and its derivative f(ϕ) 829)
      (Update bisection bounds using updated ϕ)
      (Stop the Fourier bisection iteration if converged)
      (Perform a Newton step given f(ϕ) and F(ϕ))
      (Fall back to a bisection step when ϕ is out of bounds)
      }
      (Potentially flip ϕ and return the result 830)
  }
```

Float 1062
`SampleCatmullRom()` 823
`SampleFourier()` 828

Since `SampleFourier()` operates on even functions that are periodic on the interval $[0, 2\pi]$, the probability of generating a sample in the each of the two subintervals $[0, \pi]$ and $[\pi, 2\pi]$ is equal to $1/2$. We can therefore skip the first Newton-Bisection iteration and uniformly select one of the sub-intervals with u before remapping it to the range $[0, 1]$. We then always run Newton-Bisection over $[0, \pi]$ but correct for this choice at the end of the function when the second subinterval was chosen (i.e., `flip==true`).

(Pick a side and declare bisection variables) ≡

828

```
bool flip = (u >= 0.5);
if (flip)
    u = 1 - 2 * (u - .5f);
else
    u *= 2;
double a = 0, b = Pi, phi = 0.5 * Pi;
double F, f;
```

The first fragment in the loop body of the solver evaluates the integrated $F(\phi)$ value and its derivative $f(\phi)$. Assuming a normalized function with $F(\pi) = 1$, the objective of this function is to solve an equation of the form $F(\phi) - u = 0$. In the case that F lacks proper normalization, we'd still like to generate samples proportional to the the function f , which can be achieved by adding a scaling term: $F(\phi) - uF(\pi) = 0$. The last line of the following fragment therefore subtracts u times $F(\pi)$ from F .

(Evaluate $F(\phi)$ and its derivative $f(\phi)$) ≡

828

```
<Initialize sine and cosine iterates 829>
<Initialize F and f with the first series term 830>
for (int k = 1; k < m; ++k) {
    <Compute next sine and cosine iterates 830>
    <Add the next series term to F and f 830>
}
F -= u * ak[0] * Pi;
```

As was the case in the implementation of the `Fourier()` function, it pays off to use a multiple angle formula to avoid costly trigonometric function calls when evaluating Equation (14.9):

$$\sin(k\phi) = (2 \cos\phi) \sin((k-1)\phi) - \sin((k-2)\phi). \quad [14.10]$$

Before looping over summands to compute $f(\phi)$ and $F(\phi)$, we'll initialize the initial iterates $\cos(k\phi)$ and $\sin(k\phi)$ for $k = -1$ and $k = 0$.

(Initialize sine and cosine iterates) ≡

829

```
double cosPhi = std::cos(phi);
double sinPhi = std::sqrt(1 - cosPhi * cosPhi);
double cosPhiPrev = cosPhi, cosPhiCur = 1;
double sinPhiPrev = -sinPhi, sinPhiCur = 0;
```

Fourier() 559

Pi 1063

SampleFourier() 828

The first summand of $F(\phi)$ is slightly special, so the corresponding computation for both $f(\phi)$ and $F(\phi)$ is done before the loop over the rest of the coefficients a_k .

(Initialize F and f with the first series term) ≡

829

```
F = ak[0] * phi;
f = ak[0];
```

The loop over coefficients begins by computing updated cosine and sine iterates using Equations (8.25) and (14.10).

(Compute next sine and cosine iterates) ≡

829

```
double sinPhiNext = 2 * cosPhi * sinPhiCur - sinPhiPrev;
double cosPhiNext = 2 * cosPhi * cosPhiCur - cosPhiPrev;
sinPhiPrev = sinPhiCur; sinPhiCur = sinPhiNext;
cosPhiPrev = cosPhiCur; cosPhiCur = cosPhiNext;
```

The next term of each of the sums for the function value and its derivative can now be evaluated.

(Add the next series term to F and f) ≡

829

```
F += ak[k] * recip[k] * sinPhiNext;
f += ak[k] * cosPhiNext;
```

The remaining fragments are identical to those used in `SampleCatmullRom()` for the Newton-Bisection algorithm except that the `phi` variable is used instead of `t`. We therefore won't include them here.

After `phi` has been computed, the value of the function, PDF, and azimuth angle are returned. The PDF is computed by dividing $f(\phi)$ by the normalization factor that results from $F(2\pi)$ being equal to $2\pi a_0$. As mentioned before, `phi` is flipped using the underlying symmetries when the interval $[\pi, 2\pi]$ was selected at the beginning of `SampleFourier()`.

(Potentially flip φ and return the result) ≡

828

```
if (flip)
    phi = 2 * Pi - phi;
*pdf = (Float)f / (2 * Pi * ak[0]);
*phiPtr = (Float)phi;
return f;
```

14.1.5 APPLICATION: ESTIMATING REFLECTANCE

At this point, we have covered the BxDF sampling routines of the majority of BxDFs in pbrt. As an example of their application, we will now show how these sampling routines can be used in computing estimates of the reflectance integrals defined in Section 8.1.1 for arbitrary BRDFs. For example, recall that the hemispherical-directional reflectance is

$$\rho_{hd}(\omega_o) = \int_{\mathcal{H}^2(n)} f_r(\omega_o, \omega_i) |\cos \theta_i| d\omega_i.$$

Recall from Section 8.1.1 that `BxDF::rho()` method implementations take two additional parameters, `nSamples` and an array of sample values `u`; here, we can see how they are used for Monte Carlo sampling. For BxDF implementations that can't compute the reflectance in closed form, the `nSamples` parameter specifies the number of Monte Carlo samples to take, and sample values themselves are provided in the `u` array.

BxDF 513

`BxDF::rho()` 515

Float 1062

Pi 1063

`SampleCatmullRom()` 823

`SampleFourier()` 828

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

(BxDF Method Definitions) +≡

```
Spectrum BxDF::rho(const Vector3f &w, int nSamples,
                    const Point2f *u) const {
    Spectrum r(0.);
    for (int i = 0; i < nSamples; ++i) {
        (Estimate one term of ρhd 831)
    }
    return r / nSamples;
}
```

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

$$\frac{1}{N} \sum_j^N \frac{f_r(\omega, \omega_j) |\cos \theta_j|}{p(\omega_j)}$$

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

(Estimate one term of ρ_{hd}) ≡ 831

```
Vector3f wi;
Float pdf = 0;
Spectrum f = Sample_f(w, &wi, u[i], &pdf);
if (pdf > 0) r += f * AbsCosTheta(wi) / pdf;
```

The hemispherical–hemispherical reflectance can be estimated similarly. Given

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

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

$$\frac{1}{\pi N} \sum_j^N \frac{f_r(\omega'_j, \omega''_j) |\cos \theta'_j \cos \theta''_j|}{p(\omega'_j) p(\omega''_j)}.$$

`AbsCosTheta()` 510

`BxDF` 513

`BxDF::rho()` 515

`BxDF::Sample_f()` 806

`Float` 1062

`Pi` 1063

`Point2f` 68

`Spectrum` 315

`Vector3f` 60

(BxDF Method Definitions) +≡

```
Spectrum BxDF::rho(int nSamples, const Point2f *u1,
                     const Point2f *u2) const {
    Spectrum r(0.f);
    for (int i = 0; i < nSamples; ++i) {
        (Estimate one term of ρhh 832)
    }
    return r / (Pi * nSamples);
}
```

The implementation here samples the first direction ω' uniformly over the hemisphere. Given this, the second direction can be sampled with `BxDF::Sample_f()`.⁵

(Estimate one term of ρ_{hh}) 831

```
Vector3f wo, wi;
wo = UniformSampleHemisphere(u1[i]);
Float pdf0 = UniformHemispherePdf(), pdfi = 0;
Spectrum f = Sample_f(wo, &wi, u2[i], &pdfi);
if (pdfi > 0)
    r += f * AbsCosTheta(wi) * AbsCosTheta(wo) / (pdf0 * pdfi);
```

14.1.6 SAMPLING BSDFs

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

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

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

The `BSDF::Sample_f()` method takes two random variables. The outgoing direction passed to it and the incident direction it returns are in world space.

(BSDF Method Definitions) \equiv

```
Spectrum BSDF::Sample_f(const Vector3f &woWorld, Vector3f *wiWorld,
    const Point2f &u, Float *pdf, BxDFType type,
    BxDFType *sampledType) const {
    (Choose which BxDF to sample 833)
    (Remap BxDF sample u to [0, 1)2 833)
    (Sample chosen BxDF 833)
    (Compute overall PDF with all matching BxDFs 834)
    (Compute value of BSDF for sampled direction 834)
}
```

AbsCosTheta() 510
 BSDF 572
 BSDF::Sample_f() 832
 BxDF 513
 BxDF::Sample_f() 806
 BxDFType 513
 Float 1062
 Integrator 25
 Point2f 68
 Spectrum 315
 UniformHemispherePdf() 775
 UniformSampleHemisphere() 775
 Vector3f 60

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

⁵ It could be argued that a shortcoming of the BxDF sampling interface is that there aren't entry points to sample from the 4D distribution of $f_r(p, \omega, \omega')$. This is a reasonably esoteric case for the applications envisioned for pbrt, however.

vided `BxDFType` and then uses the first dimension of the provided `u` sample to select one of the components with equal probability.

```
(Choose which BxDF to sample) ≡ 832
    int matchingComps = NumComponents(type);
    if (matchingComps == 0) {
        *pdf = 0;
        return Spectrum(0);
    }
    int comp = std::min((int)std::floor(u[0] * matchingComps),
                        matchingComps - 1);
(Get BxDF pointer for chosen component 833)
```

A second pass through the `BxDFs` is necessary to find the appropriate one that matches the flags.

```
(Get BxDF pointer for chosen component) ≡ 833
    BxDF *bxdf = nullptr;
    int count = comp;
    for (int i = 0; i < nBxDFs; ++i)
        if (bxdfs[i]->MatchesFlags(type) && count-- == 0) {
            bxdf = bxdfs[i];
            break;
    }
```

Because the `u[0]` sample was used to determine which `BxDF` component to sample, we can't directly re-use it in the call to the component's `Sample_f()` method—it's no longer uniformly distributed. (For example, if there were two matching components, then the first would only see `u[0]` values between 0 and 0.5 and the second would only see values between 0.5 and 1 if it was reused directly.) However, because `u[0]` was used to sample from a discrete distribution, we can recover a uniform random value from it: again assuming two matching components, we'd want to remap it from [0, 0.5) to [0, 1) when the first `BxDF` was sampled and from [0.5, 1) to [0, 1) when the second was. The general case of this remapping is implemented below.

```
(Remap BxDF sample u to [0, 1)2) ≡ 832
    Point2f uRemapped(u[0] * matchingComps - comp, u[1]);
```

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

```
BSDF::bxdfs 573
BSDF::LocalToWorld() 574
BSDF::NumComponents() 573
BSDF::WorldToLocal() 574
BxDF 513
BxDF::MatchesFlags() 513
BxDF::Sample_f() 806
BxDF::type 513
Point2f 68
Spectrum 315
Vector3f 60
```

```
(Sample chosen BxDF) ≡ 832
    Vector3f wi, wo = WorldToLocal(woWorld);
    *pdf = 0;
    if (sampledType *sampledType = bxdf->type;
        Spectrum f = bxdf->Sample_f(wo, &wi, uRemapped, pdf, sampledType);
        if (*pdf == 0)
            return 0;
        *wiWorld = LocalToWorld(wi);
```

To compute the actual PDF for sampling the direction ω_i , we need the average of all of the PDFs of the BxDFs that could have been used, given the BxDFType flags passed in. Because `*pdf` already holds the PDF value for the distribution the sample was taken from, we only need to add in the contributions of the others.

Given the discussion in Section 14.1.3, it's important that this step be skipped if the chosen BxDF is perfectly specular, since the PDF has an implicit delta distribution in it. It would be incorrect to add the other PDF values to this one, since it is a delta term represented with the value 1, rather than as an actual delta distribution.

```
(Compute overall PDF with all matching BxDFs) ≡ 832
if (!(bxdf->type & BSDF_SPECULAR) && matchingComps > 1)
    for (int i = 0; i < nBxDFs; ++i)
        if (bxdfs[i] != bxdf && bxdfs[i]->MatchesFlags(type))
            *pdf += bxdfs[i]->Pdf(wo, wi);
    if (matchingComps > 1) *pdf /= matchingComps;
```

Given the sampled direction, this method needs to compute the value of the BSDF for the pair of directions (ω_o, ω_i) accounting for all of the relevant components in the BSDF, unless the sampled direction was from a specular component, in which case the value returned from `Sample_f()` earlier is used. (If a specular component generated this direction, its `BxDF::f()` method will return black, even if we pass back the direction its sampling routine returned.)

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

```
(Compute value of BSDF for sampled direction) ≡ 832
if (!(bxdf->type & BSDF_SPECULAR) && matchingComps > 1) {
    bool reflect = Dot(*wiWorld, ng) * Dot(woWorld, ng) > 0;
    f = 0.;
    for (int i = 0; i < nBxDFs; ++i)
        if (bxdfs[i]->MatchesFlags(type) &&
            ((reflect && (bxdfs[i]->type & BSDF_REFLECTION)) ||
             (!reflect && (bxdfs[i]->type & BSDF_TRANSMISSION))))
            f += bxdfs[i]->f(wo, wi);
}
return f;
```

BSDF::bxdfs	573
BSDF::f()	575
BSDF::nBxDFs	573
BSDF::Pdf()	834
BSDF_ALL	513
BSDF_REFLECTION	513
BSDF_SPECULAR	513
BSDF_TRANSMISSION	513
BxDF	513
BxDF::f()	514
BxDF::MatchesFlags()	513
BxDF::Pdf()	807
BxDF::type	513
BxDFType	513
Float	1062
Vector3f	60

The `BSDF::Pdf()` method does a similar computation, looping over the BxDFs and calling their `Pdf()` methods to find the PDF for an arbitrary sampled direction. Its implementation is straightforward, so we won't include it here.

```
(BSDF Public Methods) +≡ 572
Float Pdf(const Vector3f &wo, const Vector3f &wi,
          BxDFType flags = BSDF_ALL) const;
```

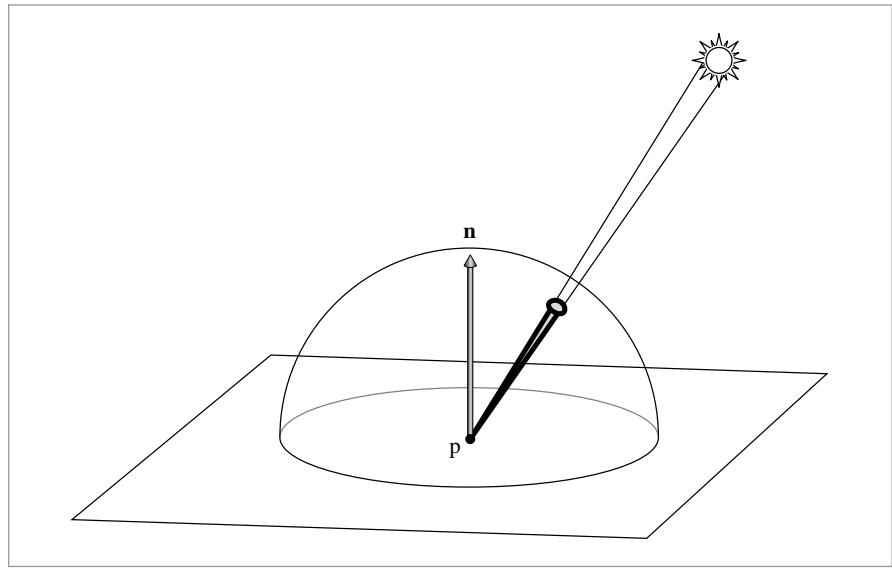


Figure 14.7: An effective sampling strategy for choosing an incident direction from a point p for direct lighting computations is to use the light source to define a distribution of directions with respect to solid angle at the point. Here, a small spherical area light source is illuminating the point. The cone of directions that the sphere subtends is a much better sampling distribution to use than a uniform distribution over the hemisphere, for example.

14.2 SAMPLING LIGHT SOURCES

Being able to take a point and sample the directions around it where direct illumination may be incident is another important sampling operation for rendering. Consider a diffuse surface illuminated by a small spherical area light source (Figure 14.7): sampling directions using the BSDF's sampling distribution is likely to be very inefficient because the light is only visible along a small cone of directions from the point. A much better approach is to instead use a sampling distribution that is based on the light source. For example, the sampling routine could choose from among only those directions where the sphere is potentially visible. This section introduces the `Light::Sample_Li()` method, which allows this operation to be implemented for the lights in pbrt.

There are two sampling methods that `Lights` must implement. The first, `Sample_Li()`, samples an incident direction at a point in the scene along which illumination from the light may be arriving. The second, `Light::Sample_Le()`, will be defined in Section 16.1.2; it returns a light-carrying ray leaving the light source. Both have corresponding methods that return the PDF for an incident direction or a ray, respectively.

`Light` 714

`Light::Sample_Le()` 955

`Light::Sample_Li()` 716

The `Light::Sample_Li()` method was first introduced in Section 12.2. For reference, here is its declaration:

```
virtual Spectrum Sample_Li(const Interaction &ref, const Point2f &u,
                           Vector3f *wi, Float *pdf, VisibilityTester *vis) const = 0;
```

We can now understand the meaning of its `u` and `pdf` parameters: `u` provides a 2D sample value for sampling the light source, and the PDF for sampling the chosen direction is returned in `*pdf`.

The Light's `Pdf_Li()` method returns the probability density with respect to solid angle for the light's `Sample_Li()` method to sample the direction `wi` from the reference point `ref`.

(Light Interface) +≡

714

```
virtual Float Pdf_Li(const Interaction &ref,
                      const Vector3f &wi) const = 0;
```

14.2.1 LIGHTS WITH SINGULARITIES

Just as with perfect specular reflection and transmission, light sources that are defined in terms of delta distributions fit naturally into this sampling framework, although they require care on the part of the routines that call their sampling methods, since there are implicit delta distributions in the radiance and PDF values that they return. For the most part, these delta distributions naturally cancel out when estimators are evaluated, although multiple importance sampling code must be aware of this case, just as was the case with BSDFs. (The `IsDeltaLight()` utility function can be used to see if a light source is described by a delta distribution.)

Point lights are described by a delta distribution such that they only illuminate a receiving point from a single direction. Thus, the sampling problem is trivial. The `PointLight::Sample_Li()` method was already implemented in Section 12.3, where we glossed over the nuance that the method performs Monte Carlo sampling of a delta distribution and thus always returns a single direction and doesn't need the random sample values.

Due to the delta distribution, the `PointLight::Pdf_Li()` method returns 0. This value reflects the fact that there is no chance for some other sampling process to randomly generate a direction that would intersect an infinitesimal light source.

(PointLight Method Definitions) +≡

```
Float PointLight::Pdf_Li(const Interaction &, const Vector3f &) const {
    return 0;
}
```

The `Sample_Li()` methods for `SpotLights`, `ProjectionLights`, `GonioPhotometricLights`, and `DistantLights` were also previously implemented in Sections 12.3.1, 12.3.2, 12.3.3, and 12.4, respectively. All also return 0 from their `Pdf_Li()` methods.

14.2.2 SAMPLING SHAPES

In pbrt, area lights are defined by attaching an emission profile to a `Shape`. Therefore, in order to sample incident illumination from such light sources, it is useful to be able to generate samples over the surface of shapes. To make this possible, we will add sampling methods to the `Shape` class that sample points on their surfaces. The `AreaLight` sampling methods to be defined shortly will in turn call these methods.

There are two shape sampling methods, both named `Shape::Sample()`. The first chooses points on the surface of the shape using a sampling distribution with respect to sur-

AreaLight 734
DistantLight 731
Float 1062
GonioPhotometricLight 728
Interaction 115
IsDeltaLight() 715
Light 714
PointLight 719
PointLight::Sample_Li() 721
ProjectionLight 724
Shape 123
Shape::Sample() 837
SpotLight 721
Vector3f 60

face area and returns the local geometric information about the sampled point in an `Interaction`. In addition to initializing the position `p` and normal `n` of the sampled point, the `Sample()` method should set `Interaction::pError` with bounds on the floating-point rounding error in the computed `p` value. `pError` is used to compute robust origins for rays leaving the surface of the light—see Section 3.9.5.

```
<Shape Interface> +≡
    virtual Interaction Sample(const Point2f &u) const = 0;
```

123

Shapes almost always sample uniformly by area on their surface. Therefore, we will provide a default implementation of the `Shape::Pdf()` method corresponding to this sampling approach that returns the corresponding PDF: 1 over the surface area.

```
<Shape Interface> +≡
    virtual Float Pdf(const Interaction &) const {
        return 1 / Area();
    }
```

123

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

```
<Shape Interface> +≡
    virtual Interaction Sample(const Interaction &ref,
                               const Point2f &u) const {
        return Sample(u);
    }
```

123

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

```
Float 1062
Interaction 115
Interaction::pError 115
Point2f 68
Shape 123
Shape::Area() 131
Shape::Pdf() 837
Shape::Sample() 837
Vector3f 60

<Shape Method Definitions> +≡
    Float Shape::Pdf(const Interaction &ref,
                      const Vector3f &wi) const {
        <Intersect sample ray with area light geometry 838>
        <Convert light sample weight to solid angle measure 838>
        return pdf;
    }
```

Given a reference point and direction ω_i , the `Pdf()` method determines if the ray from the point in direction ω_i intersects the shape. If the ray doesn't intersect the shape at all, the probability that the shape would have chosen the direction ω_i can be assumed to be 0

(an effective sampling algorithm wouldn't generate such a sample, and in any case the light will not contribute to such directions, so using a zero probability density is fine).

Note that this ray intersection test is only between the ray and the single shape under consideration. The rest of the scene geometry is ignored, and thus the intersection test is fairly efficient.

(Intersect sample ray with area light geometry) ≡

837

```
Ray ray = ref.SpawnRay(wi);
Float tHit;
SurfaceInteraction isectLight;
if (!Intersect(ray, &tHit, &isectLight, false)) return 0;
```

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

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

where θ_o is the angle between the direction of the ray from the point on the light to the reference point and the light's surface normal, and r^2 is the distance between the point on the light and the point being shaded (recall the discussion about transforming between area and directional integration domains in Section 5.5).

(Convert light sample weight to solid angle measure) ≡

837

```
Float pdf = DistanceSquared(ref.p, isectLight.p) /
    (AbsDot(isectLight.n, -wi) * Area());
```

Sampling Disks

The Disk sampling method uses the concentric disk sampling function to find a point on the unit disk and then scales and offsets this point to lie on the disk of a given radius and height. Note that this method does not account for partial disks due to `Disk::innerRadius` being nonzero or `Disk::phiMax` being less than 2π . Fixing this bug is left for an exercise at the end of the chapter.

Because the object space z value of the sampled point is equal to `Disk::height`, zero-extent bounds can be used for the error bounds for rays leaving the sampled point, just as with ray–disk intersections. (These bounds may later be expanded by the object to world transformation, however.)

(Disk Method Definitions) +≡

```
Interaction Disk::Sample(const Point2f &u) const {
    Point2f pd = ConcentricSampleDisk(u);
    Point3f p0bj(pd.x * radius, pd.y * radius, height);
    Interaction it;
    it.n = Normalize((*ObjectToWorld)(Normal3f(0, 0, 1)));
    if (reverseOrientation) it.n *= -1;
    it.p = (*ObjectToWorld)(p0bj, Vector3f(0, 0, 0), &it.pError);
    return it;
}
```

AbsDot() 64
 ConcentricSampleDisk() 778
 Disk 146
 Disk::height 147
 Disk::innerRadius 147
 Disk::phiMax 147
 Disk::radius 147
 DistanceSquared() 70
 Float 1062
 Interaction 115
 Interaction::n 116
 Interaction::p 115
 Interaction::pError 115
 Interaction::SpawnRay() 232
 Normal3f 71
 Point2f 68
 Point3f 68
 Ray 73
 Shape::Area() 131
 Shape::Intersect() 129
 Shape::ObjectToWorld 124
 Shape::reverseOrientation
 124
 SurfaceInteraction 116
 Vector3f::Normalize() 66
 Vector3f 60

Sampling Cylinders

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

(Cylinder Method Definitions) +≡

```
Interaction Cylinder::Sample(const Point2f &u) const {
    Float z = Lerp(u[0], zMin, zMax);
    Float phi = u[1] * phiMax;
    Point3f p0bj = Point3f(radius * std::cos(phi), radius * std::sin(phi),
                           z);
    Interaction it;
    it.n = Normalize((*ObjectToWorld)(Normal3f(p0bj.x, p0bj.y, 0)));
    if (reverseOrientation) it.n *= -1;
    <Reproject p0bj to cylinder surface and compute p0bjError 839>
    it.p = (*ObjectToWorld)(p0bj, p0bjError, &it.pError);
    return it;
}
```

If the system's `std::sin()` and `std::cos()` functions compute results that are as precise as possible—i.e., they always return the closest floating-point value to the fully-precise result, then it can be shown that the x and y components of `p0bj` are within a factor of γ_3 of the actual surface of the cylinder. While many implementations of those functions are that precise, not all are, especially on GPUs. To be safe, the implementation here reprojects the sampled point to lie on the cylinder. In this case, the error bounds are the same as were derived for reprojected ray–cylinder intersection points in Section 3.9.4.

Cylinder::phiMax 143
 Cylinder::radius 143
 Cylinder::zMax 143
 Cylinder::zMin 143
 Float 1062
 gamma() 217
 Interaction 115
 Interaction::n 116
 Interaction::p 115
 Interaction::pError 115
 Lerp() 1079
 Normal3f 71
 Point2f 68
 Point3f 68
 Shape::ObjectToWorld 124
 Shape::reverseOrientation 124
 Triangle 156
 UniformSampleTriangle() 782
 Vector3f::Normalize() 66
 Vector3f 60

(Reproject p0bj to cylinder surface and compute p0bjError) ≡

839

```
Float hitRad = std::sqrt(p0bj.x * p0bj.x + p0bj.y * p0bj.y);
p0bj.x *= radius / hitRad;
p0bj.y *= radius / hitRad;
Vector3f p0bjError = gamma(3) * Abs(Vector3f(p0bj.x, p0bj.y, 0));
```

Sampling Triangles

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

(Triangle Method Definitions) +≡

```
Interaction Triangle::Sample(const Point2f &u) const {
    Point2f b = UniformSampleTriangle(u);
    <Get triangle vertices in p0, p1, and p2 157>
    Interaction it;
    it.p = b[0] * p0 + b[1] * p1 + (1 - b[0] - b[1]) * p2;
    <Compute surface normal for sampled point on triangle 840>
    <Compute error bounds for sampled point on triangle 840>
    return it;
}
```

```
(Compute surface normal for sampled point on triangle) ≡ 839
if (mesh->n)
    it.n = Normalize(b[0] * mesh->n[v[0]] +
                      b[1] * mesh->n[v[1]] +
                      (1 - b[0] - b[1]) * mesh->n[v[2]]);
else
    it.n = Normalize(Normal3f(Cross(p1 - p0, p2 - p0)));
if (reverseOrientation) it.n *= -1;
```

Because the sampled point is computed with barycentric interpolation, it has the same error bounds as were computed in Section 3.9.4 for triangle intersection points.

```
(Compute error bounds for sampled point on triangle) ≡ 839
Point3f pAbsSum = Abs(b[0] * p0) + Abs(b[1] * p1) +
                  Abs((1 - b[0] - b[1]) * p2);
it.pError = gamma(6) * Vector3f(pAbsSum);
```

Sampling Spheres

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

(Sphere Method Definitions) +≡

```
Interaction Sphere::Sample(const Point2f &u) const {
    Point3f pObj = Point3f(0, 0, 0) + radius * UniformSampleSphere(u);
    Interaction it;
    it.n = Normalize((*ObjectToWorld)(Normal3f(pObj.x, pObj.y, pObj.z)));
    if (reverseOrientation) it.n *= -1;
    (Reproject pObj to sphere surface and compute pObjError 840)
    it.p = (*ObjectToWorld)(pObj, pObjError, &it.pError);
    return it;
}
```

Because `UniformSampleSphere()` uses `std::sin()` and `std::cos()`, the error bounds on the computed `pObj` value depend on the accuracy of those functions. Therefore, as with cylinders, the sampled point is reprojected to the sphere's surface, so that the error bounds derived earlier in Equation (3.14) can be used without needing to worry about those functions' accuracy.

(Reproject pObj to sphere surface and compute pObjError) ≡ 840, 844

```
pObj *= radius / Distance(pObj, Point3f(0, 0, 0));
Vector3f pObjError = gamma(5) * Abs((Vector3f)pObj);
```

For the sphere sampling method that is given a point being illuminated, we can do much better than sampling over the sphere's entire area. While uniform sampling over its surface is perfectly correct, a better approach is to not sample points on the sphere that are definitely not visible from the point (such as those on the back side of the sphere as seen from the point). The sampling routine here instead uniformly samples directions

Cross() 65
 Disk 146
 Distance() 70
 gamma() 217
 Interaction 115
 Interaction:: 116
 Interaction::p 115
 Interaction::pError 115
 Normal3::Normalize() 71
 Normal3f 71
 Point2f 68
 Point3::Abs() 71
 Point3f 68
 Shape::ObjectToWorld 124
 Shape::reverseOrientation 124
 Sphere::radius 133
 Triangle::mesh 156
 Triangle::v 156
 TriangleMesh::n 155
 UniformSampleSphere() 776
 Vector3::Abs() 63
 Vector3::Normalize() 66
 Vector3f 60

over the solid angle subtended by the sphere from the reference point and then computes the point on the sphere corresponding to the sampled direction.

(Sphere Method Definitions) +≡

```
Interaction Sphere::Sample(const Interaction &ref,
    const Point2f &u) const {
    (Compute coordinate system for sphere sampling 841)
    (Sample uniformly on sphere if p is inside it 841)
    (Sample sphere uniformly inside subtended cone 841)
}
```

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

(Compute coordinate system for sphere sampling) ≡

841

```
Point3f pCenter = (*ObjectToWorld)(Point3f(0, 0, 0));
Vector3f wc = Normalize(pCenter - ref.p);
Vector3f wcX, wcY;
CoordinateSystem(wc, &wcX, &wcY);
```

For points that lie inside the surface of the sphere, the entire sphere should be sampled, since the whole sphere is visible from inside it. Note that the reference point used in this determination, p_{origin} , is computed using the `OffsetRayOrigin()` function. Doing so ensures that if the reference point came from a ray intersecting the sphere, the point tested lies on the correct side of the sphere.

(Sample uniformly on sphere if p is inside it) ≡

841

```
Point3f pOrigin = OffsetRayOrigin(ref.p, ref.pError, ref.n,
    pCenter - ref.p);
if (DistanceSquared(pOrigin, pCenter) <= radius * radius)
    return Sample(u);
```

Otherwise sampling within the cone proceeds.

(Sample sphere uniformly inside subtended cone) ≡

841

```
<Compute  $\theta$  and  $\phi$  values for sample in cone 842>
<Compute angle  $\alpha$  from center of sphere to sampled point on surface 844>
<Compute surface normal and sampled point on sphere 844>
<Return Interaction for sampled point on sphere 844>
```

If the reference point is outside the sphere, then as seen from the point being shaded p the sphere subtends an angle

$$\theta_{\max} = \arcsin \left(\frac{r}{|p - p_c|} \right) = \arccos \sqrt{1 - \left(\frac{r}{|p - p_c|} \right)^2}, \quad (14.11)$$

where r is the radius of the sphere and p_c is its center (Figure 14.8). The sampling method here computes the cosine of the subtended angle θ_{\max} using Equation (14.11) and then uniformly samples directions inside this cone of directions using the approach that was derived earlier for the `UniformSampleCone()` function, sampling an offset θ from the center vector ω_c and then uniformly sampling a rotation angle ϕ around the vector.

CoordinateSystem() 67
 DistanceSquared() 70
 Interaction 115
 Interaction::n 116
 Interaction::p 115
 Interaction::pError 115
 OffsetRayOrigin() 231
 Point2f 68
 Point3f 68
 Shape::ObjectToWorld 124
 Sphere 133
 Sphere::radius 133
 Sphere::Sample() 840
 UniformSampleCone() 781
 Vector3f::Normalize() 66
 Vector3f 60

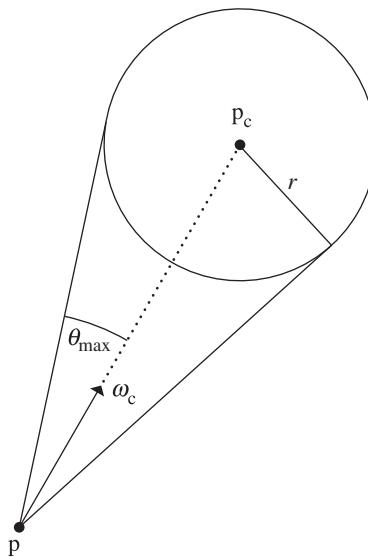


Figure 14.8: To sample points on a spherical light source, we can uniformly sample within the cone of directions around a central vector ω_c with an angular spread of up to θ_{\max} . Trigonometry can be used to derive the value of $\sin \theta_{\max}, r/|\mathbf{p}_c - \mathbf{p}|$.

That function isn't used here, however, as we will need some of the intermediate values in the following fragments.

(Compute θ and ϕ values for sample in cone) ≡

841

```
Float sinThetaMax2 = radius * radius / DistanceSquared(ref.p, pCenter);
Float cosThetaMax = std::sqrt(std::max((Float)0, 1 - sinThetaMax2));
Float cosTheta = (1 - u[0]) + u[0] * cosThetaMax;
Float sinTheta = std::sqrt(std::max((Float)0, 1 - cosTheta * cosTheta));
Float phi = u[1] * 2 * Pi;
```

Given a sample angle (θ, ϕ) with respect to the sampling coordinate system computed earlier, we can directly compute the corresponding sample point on the sphere. The derivation of this approach follows three steps, illustrated in Figure 14.9.

First, if we denote the distance from the reference point to the center of the sphere by d_c and form a right triangle with angle θ at the reference point, then we can see that the lengths of the other two sides of the triangle, as shown in Figure 14.9(a), are $d_c \cos \theta$ and $d_c \sin \theta$.

Next, consider the right triangle shown in Figure 14.9(b), where the hypotenuse is the line segment with length equal to the sphere's radius r that goes from the center of the sphere to the point where the line from the sampled angle intersects the sphere. From the Pythagorean theorem, we can see that the length of the third side of that triangle is

$$\sqrt{r^2 - d_c^2 \sin^2 \theta}.$$

DistanceSquared() 70

Float 1062

Interaction::p 115

Pi 1063

Sphere::radius 133

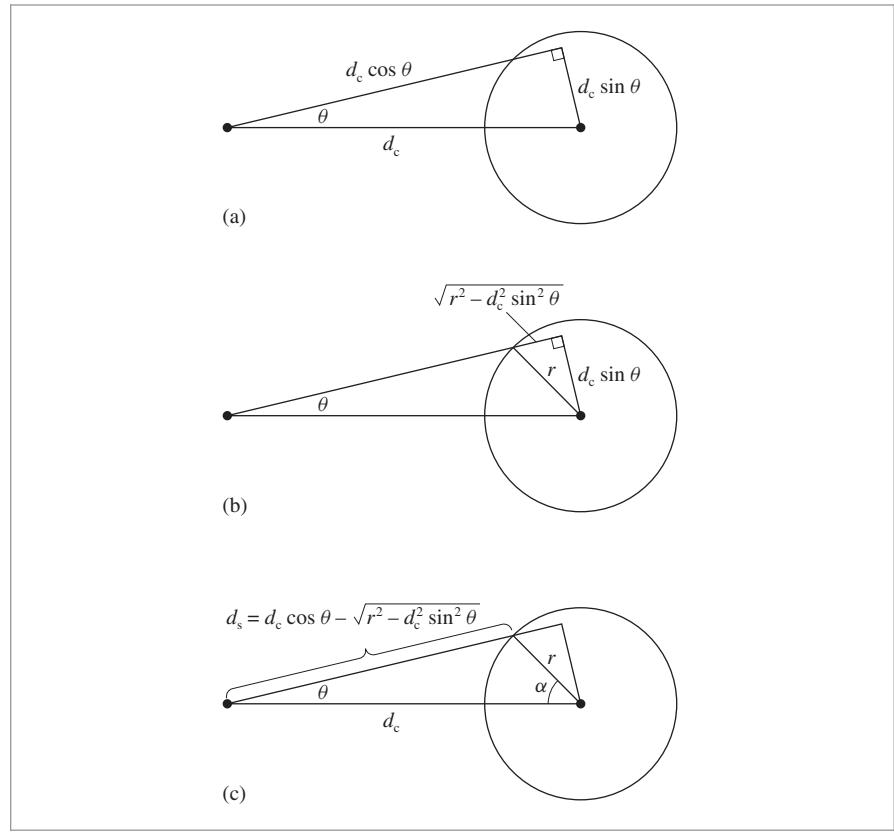


Figure 14.9: Geometric Setting for Computing the Sampled Point on the Sphere Corresponding to a Sampled Angle θ . (a) Right triangle with hypotenuse d_c and angle θ . (b) The right triangle with hypotenuse from the center of the sphere to the sampled point on the sphere. (c) Given the lengths of the three sides of this triangle, the law of cosines gives the angle α to the sampled point.

Subtracting this length from $d_c \cos \theta$ gives us the length of the segment from the reference point to the sampled point on the sphere:

$$d_s = d_c \cos \theta - \sqrt{r^2 - d_c^2 \sin^2 \theta}.$$

We can now compute the angle α from the center of the sphere to the sampled point on the sphere using the law of cosines, which relates the squared lengths of two sides of the triangle and the angle α opposite them:

$$d_s^2 = d_c^2 + r^2 - 2d_c r \cos \alpha.$$

(See Figure 14.9(c).) Solving for $\cos \alpha$, we have

$$\cos \alpha = \frac{d_c^2 + r^2 - d_s^2}{2d_c r}.$$

(Compute angle α from center of sphere to sampled point on surface) ≡ 841

```
Float dc = Distance(ref.p, pCenter);
Float ds = dc * cosTheta -
    std::sqrt(std::max((Float)0,
                       radius * radius - dc * dc * sinTheta * sinTheta));
Float cosAlpha = (dc * dc + radius * radius - ds * ds) /
    (2 * dc * radius);
Float sinAlpha = std::sqrt(std::max((Float)0, 1 - cosAlpha * cosAlpha));
```

The α angle and ϕ give the spherical coordinates for the sampled point on the unit sphere, with respect to a coordinate system centered around the vector from the sphere center to the reference point. Since we earlier computed a coordinate system from the reference point to the center, we can use that one here with each vector flipped.

(Compute surface normal and sampled point on sphere) ≡ 841

```
Vector3f nObj = SphericalDirection(sinAlpha, cosAlpha, phi,
    -wcX, -wcY, -wc);
Point3f pObj = radius * Point3f(nObj.x, nObj.y, nObj.z);
```

As with the other `Sphere::Sample()` method, the sampled point is reprojected onto the surface of the sphere; in turn, we can use the same error bounds for the computed point as were derived earlier.

(Return Interaction for sampled point on sphere) ≡ 841

```
Interaction it;
(Reproject pObj to sphere surface and compute pObjError 840)
it.p = (*ObjectToWorld)(pObj, pObjError, &it.pError);
it.n = (*ObjectToWorld)(Normal3f(nObj));
if (reverseOrientation) it.n *= -1;
return it;
```

The value of the PDF for sampling a direction toward a sphere from a given point depends on which of the two sampling strategies would be used for the point.

(Sphere Method Definitions) +≡

```
Float Sphere::Pdf(const Interaction &ref, const Vector3f &wi) const {
    Point3f pCenter = (*ObjectToWorld)(Point3f(0, 0, 0));
    (Return uniform PDF if point is inside sphere 844)
    (Compute general sphere PDF 845)
}
```

If the reference point was inside the sphere, a uniform sampling strategy was used, in which case, the implementation hands off to the `Pdf()` method of the `Shape` class, which takes care of the solid angle conversion.

(Return uniform PDF if point is inside sphere) ≡ 844

```
Point3f pOrigin = OffsetRayOrigin(ref.p, ref.pError, ref.n,
    pCenter - ref.p);
if (DistanceSquared(pOrigin, pCenter) <= radius * radius)
    return Shape::Pdf(ref, wi);
```

Distance() [70](#)
 DistanceSquared() [70](#)
 Float [1062](#)
 Interaction [115](#)
 Interaction::n [116](#)
 Interaction::p [115](#)
 Interaction::pError [115](#)
 Normal3f [71](#)
 OffsetRayOrigin() [231](#)
 Point3f [68](#)
 Shape [123](#)
 Shape::ObjectToWorld [124](#)
 Shape::Pdf() [837](#)
 Shape::reverseOrientation
[124](#)
 Sphere::radius [133](#)
 SphericalDirection() [346](#)
 Vector3f [60](#)

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

(Compute general sphere PDF) \equiv 844

```
Float sinThetaMax2 = radius * radius / DistanceSquared(ref.p, pCenter);
Float cosThetaMax = std::sqrt(std::max((Float)0, 1 - sinThetaMax2));
return UniformConePdf(cosThetaMax);
```

14.2.3 AREA LIGHTS

Given shape sampling methods, the `DiffuseAreaLight::Sample_Li()` method is quite straightforward. Most of the hard work is done by the Shapes, and the `DiffuseAreaLight` just needs to copy appropriate values to output parameters and compute the emitted radiance value.

(DiffuseAreaLight Method Definitions) \equiv

```
Spectrum DiffuseAreaLight::Sample_Li(const Interaction &ref,
    const Point2f &u, Vector3f *wi, Float *pdf,
    VisibilityTester *vis) const {
    Interaction pShape = shape->Sample(ref, u);
    pShape.mediumInterface = mediumInterface;
    *wi = Normalize(pShape.p - ref.p);
    *pdf = shape->Pdf(ref, *wi);
    *vis = VisibilityTester(ref, pShape);
    return L(pShape, -*wi);
}
```

[DiffuseAreaLight](#) 736
[DiffuseAreaLight::L\(\)](#) 736
[DiffuseAreaLight::Sample_Li\(\)](#) 845
[DiffuseAreaLight::shape](#) 736
[Float](#) 1062
[InfiniteAreaLight](#) 737
[Interaction](#) 115
[Interaction::mediumInterface](#) 116
[Interaction::p](#) 115
[Point2f](#) 68
[Shape](#) 123
[Shape::Pdf\(\)](#) 837
[Shape::Sample\(\)](#) 837
[Spectrum](#) 315
[Sphere::radius](#) 133
[UniformConePdf\(\)](#) 781
[Vector3f::Normalize\(\)](#) 66
[Vector3f](#) 60
[VisibilityTester](#) 717

`Pdf_Li()` calls the variant of `Shape::Pdf()` that returns a density with respect to solid angle, so the value it returns from can be returned directly.

(DiffuseAreaLight Method Definitions) \equiv

```
Float DiffuseAreaLight::Pdf_Li(const Interaction &ref,
    const Vector3f &wi) const {
    return shape->Pdf(ref, wi);
}
```

14.2.4 INFINITE AREA LIGHTS

The `InfiniteAreaLight`, defined in Section 12.6, can be considered to be an infinitely large sphere that surrounds the entire scene, illuminating it from all directions. The environment maps used with `InfiniteAreaLights` often have substantial variation along different directions: consider, for example, an environment map of the sky during daytime, where the relatively small number of directions that the sun subtends will be thousands of times brighter than the rest of the directions. Given this substantial variation, implementing a sampling method for `InfiniteAreaLights` that matches the illumination distribution will generally substantially reduce variance in images.

Figure 14.10 shows two images of a car model illuminated by the morning skylight environment map from Figure 12.21. The top image was rendered using a simple cosine-weighted sampling distribution for selecting incident illumination directions, while the



(a)



(b)

Figure 14.10: Car Model Illuminated by the Morning Skylight Environment Map. Both images were rendered with four image samples per pixel and eight light source samples per image sample. (a) The result of using a uniform sampling distribution and (b) the improvement from the importance sampling method implemented here. A total of just 32 light samples per pixel gives an excellent result with this approach.

bottom image was rendered using the sampling method implemented in this section. Both images used just 32 shadow samples per pixel. For the same number of samples taken and with negligible additional computational cost, this sampling method computes a much better result with much lower variance.

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

1. We define a piecewise-constant 2D probability distribution function in (u, v) image coordinates $p(u, v)$ that corresponds to the distribution of the radiance represented by the environment map.

2. We apply the sampling method from Section 13.6.7 to transform 2D samples to samples drawn from the piecewise-constant $p(u, v)$ distribution.
3. We define a probability density function over directions on the unit sphere based on the probability density over (u, v) .

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

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

(Initialize sampling PDFs for infinite area light) ≡
(Compute scalar-valued image img from environment map 848)
(Compute sampling distributions for rows and columns of image 848)

739

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

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

The second thing of note in this code is that the piecewise constant function values being stored here in `img` are found by slightly blurring the radiance function with the `MIPMap::Lookup()` method (rather than just copying the corresponding texel values). The motivation for this is subtle; Figure 14.11 illustrates the idea in 1D. Because the continuous radiance function used for rendering is reconstructed by bilinearly interpolating between texels in the image, just because some texel is completely black, for example, the radiance function may be nonzero a tiny distance away from it due to a neighboring texel's contribution. Because we are using a piecewise-constant function for sampling rather than a piecewise-linear one, it must account for this issue in order to ensure greater-than-zero probability of sampling any point where the radiance function is nonzero.⁶

Finally, each image value in the `img` buffer is multiplied by the value of $\sin \theta$ corresponding to the θ value each row has when the latitude-longitude image is mapped to the sphere. Note that this multiplication has no effect on the sampling method's correctness: because the value of $\sin \theta$ is always greater than 0 over the $[0, \pi]$ range, we are just reshaping the sampling PDF. The motivation for adjusting the PDF is to eliminate the

`InfiniteAreaLight` 737

`Spectrum::y()` 325

⁶ Alternatively, we could use a piecewise-linear function for importance sampling and thus match the radiance function exactly. However, it's easier to draw samples from a piecewise-constant function's distribution, and, because environment maps generally have a large number of texel samples, piecewise-constant functions generally suffice to match their distributions well. See Exercise 14.5 for discussion about handling this issue more robustly.

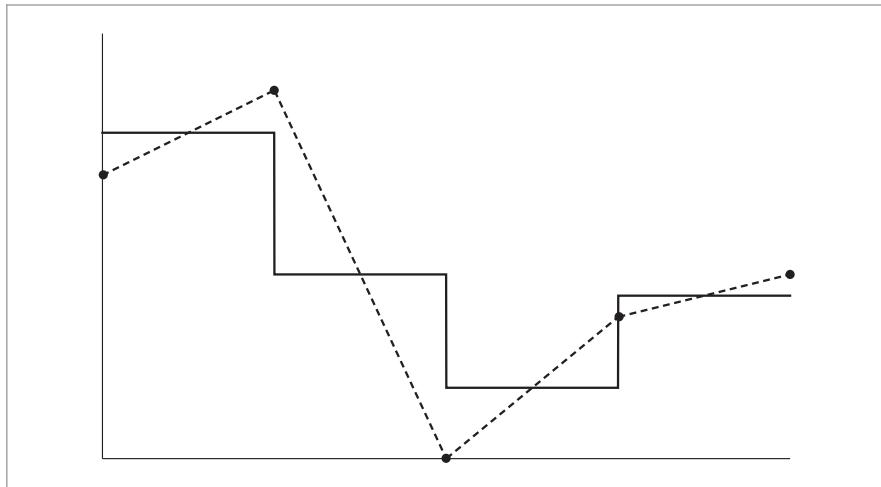


Figure 14.11: Finding a Piecewise-Constant Function (solid lines) That Approximates a Piecewise Linear Function (dashed lines) for Use as a Sampling Distribution. Even though some of the sample points that define the piecewise linear function (solid dots) may be zero-valued, the piecewise-constant sampling function must not be zero over any range where the actual function is nonzero. A reasonable approach to avoid this case, shown here and implemented in the `InfiniteAreaLight` sampling routines, is to find the average value of the function over some range and use that to define the piecewise-constant function.

effect of the distortion from mapping the 2D image to the unit sphere in the sampling method here; the details will be fully explained later in this section.

```
(Compute scalar-valued image img from environment map) ≡ 847
    int width = resolution.x, height = resolution.y;
    Float filter = (Float)1 / std::max(width, height);
    std::unique_ptr<Float[]> img(new Float[width * height]);
    for (int v = 0; v < height; ++v) {
        Float vp = (Float)v / (Float)height;
        Float sinTheta = std::sin(Pi * Float(v + .5f) / Float(height));
        for (int u = 0; u < width; ++u) {
            Float up = (Float)u / (Float)width;
            img[u + v * width] = Lmap->Lookup(Point2f(up, vp), filter).y();
            img[u + v * width] *= sinTheta;
        }
    }
}
```

`Distribution2D` 785

`Float` 1062

`InfiniteAreaLight::distribution`
848

`InfiniteAreaLight::Lmap` 740

`MIPMap::Lookup()` 635

`Pi` 1063

`Point2f` 68

`Spectrum::y()` 325

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

```
(Compute sampling distributions for rows and columns of image) ≡ 847
    distribution.reset(new Distribution2D(img.get(), width, height));
```

```
(InfiniteAreaLight Private Data) +≡ 737
    std::unique_ptr<Distribution2D> distribution;
```

Given this precomputed data, the task of the sampling method is relatively straightforward. Given a sample u over $[0, 1]^2$, it draws a sample from the function's distribution using the sampling algorithm described in Section 13.6.7, which gives a (u, v) value and the value of the PDF for taking this sample, $p(u, v)$.

```
<InfiniteAreaLight Method Definitions> +≡
Spectrum InfiniteAreaLight::Sample_Li(const Interaction &ref,
    const Point2f &u, Vector3f *wi, Float *pdf,
    VisibilityTester *vis) const {
    <Find (u, v) sample coordinates in infinite light texture 849>
    <Convert infinite light sample point to direction 849>
    <Compute PDF for sampled infinite light direction 850>
    <Return radiance value for infinite light direction 850>
}
```

```
<Find (u, v) sample coordinates in infinite light texture> ≡
    849
    Float mapPdf;
    Point2f uv = distribution->SampleContinuous(u, &mapPdf);
    if (mapPdf == 0) return Spectrum(0.f);
```

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

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

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

```
<Convert infinite light sample point to direction> ≡
849
    Float theta = uv[1] * Pi, phi = uv[0] * 2 * Pi;
    Float cosTheta = std::cos(theta), sinTheta = std::sin(theta);
    Float sinPhi = std::sin(phi), cosPhi = std::cos(phi);
    *wi = LightToWorld(Vector3f(sinTheta * cosPhi, sinTheta * sinPhi,
        cosTheta));
```

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

```
Distribution2D::
    SampleContinuous()
786
Float 1062
Interaction 115
Light::LightToWorld 715
Pi 1063
Point2f 68
Spectrum 315
Vector3f 60
VisibilityTester 717
```

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

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

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

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

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

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

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

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

(Compute PDF for sampled infinite light direction) ≡

849

```
*pdf = mapPdf / (2 * Pi * Pi * sinTheta);
if (sinTheta == 0) *pdf = 0;
```

Distribution2D::Pdf() 787

Float 1062

InfiniteAreaLight::
distribution
848

InfiniteAreaLight::Lmap 740

InfiniteAreaLight::Pdf_Li()
850

InfiniteAreaLight::
worldRadius
740

Interaction 115

Interaction::p 115

Interaction::time 115

Inv2Pi 1063

InvPi 1063

Light::mediumInterface 715

Light::WorldToLight 715

MIPMap::Lookup() 635

Pi 1063

Point2f 68

Spectrum 315

SpectrumType::Illuminant 330

SphericalPhi() 346

SphericalTheta() 346

Vector3f 60

VisibilityTester 717

The method can finally initialize the `VisibilityTester` with a light sample point outside the scene's bounds and return the radiance value for the chosen direction.

(Return radiance value for infinite light direction) ≡

849

```
*vis = VisibilityTester(ref, Interaction(ref.p + *wi * (2 * worldRadius),
                                         ref.time, mediumInterface));
return Spectrum(Lmap->Lookup(uv), SpectrumType::Illuminant);
```

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

(InfiniteAreaLight Method Definitions) +=

```
Float InfiniteAreaLight::Pdf_Li(const Interaction &,
                                const Vector3f &w) const {
    Vector3f wi = WorldToLight(w);
    Float theta = SphericalTheta(wi), phi = SphericalPhi(wi);
    Float sinTheta = std::sin(theta);
    if (sinTheta == 0) return 0;
    return distribution->Pdf(Point2f(phi * Inv2Pi, theta * InvPi)) /
           (2 * Pi * Pi * sinTheta);
}
```



Figure 14.12: 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.*)

14.3 DIRECT LIGHTING

Before we introduce the light transport equation in its full generality, we will implement the `DirectLightingIntegrator` which, unsurprisingly, only 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, except for basic specular reflection and transmission effects. Starting out with this integrator allows us to focus on some of the important 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 14.12 shows the San Miguel scene rendered with direct lighting only.

```
<DirectLightingIntegrator Declarations> ≡
    class DirectLightingIntegrator : public SamplerIntegrator {
        public:
            <DirectLightingIntegrator Public Methods>
        private:
            <DirectLightingIntegrator Private Data 852>
    };
}
```

`Light:::nSamples` 715
`SamplerIntegrator` 26

The implementation provides two different strategies for computing direct lighting. Each computes an unbiased estimate of exitant radiance at a point in a given direction. The `LightStrategy` enumeration records which approach has been selected. The first strategy, `UniformSampleAll`, loops over all of the lights and takes a number of samples based on `Light:::nSamples` from each of them, summing the result. (Recall the discussion of

splitting in Section 13.7.1—the `UniformSampleAll` strategy is applying this technique.) The second, `UniformSampleOne`, takes a single sample from just one of the lights, chosen at random.

(LightStrategy Declarations) ≡

```
enum class LightStrategy { UniformSampleAll, UniformSampleOne };
```

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., to resolve depth of field without excessive noise), then a single light sample per image sample may be more appropriate: in the aggregate all of the image samples in a pixel will sample the direct lighting well enough to give a high-quality image. Alternatively, if few samples per pixel are being taken, sampling all lights may be preferable to ensure a noise-free result.

The `DirectLightingIntegrator` constructor, which we won’t include here, just passes a `Camera` and `Sampler` to the `SamplerIntegrator` base class constructor and initializes two member variables. In addition to the direct lighting strategy, the `DirectLightingIntegrator` stores a maximum recursion depth for rays that are traced to account for specular reflection or specular transmission.

(DirectLightingIntegrator Private Data) ≡

851

```
const LightStrategy strategy;
const int maxDepth;
```

The numbers 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 two two-dimensional samples obtained via `Sampler::Get2D()` suffice—one for selecting a position on a light source and one for sampling a scattered direction from the BSDF.

When multiple samples are taken per light, the integrator requests sample arrays from the sampler before the main rendering process begins. Using sample arrays is preferable to performing many separate calls to `Sampler::Get2D()` because it gives the sampler an opportunity to use improved sample placement techniques, optimizing the distribution of samples over the entire array and thus, over all of the light source samples for the current point being shaded.

(DirectLightingIntegrator Method Definitions) ≡

```
void DirectLightingIntegrator::Preprocess(const Scene &scene,
    Sampler &sampler) {
    if (strategy == LightStrategy::UniformSampleAll) {
        <Compute number of samples to use for each light 853>
        <Request samples for sampling all lights 853>
    }
}
```

BSDF 572
 Camera 356
`DirectLightingIntegrator::strategy` 852
`Light::nSamples` 715
`LightStrategy` 852
`LightStrategy::UniformSampleAll` 852
`Sampler` 421
`Sampler::Get2D()` 422
`Sampler::RoundCount()` 424
`SamplerIntegrator` 25
`Scene` 23

For the `LightStrategy::UniformSampleAll` strategy, each light stores a desired number of samples in its `Light::nSamples` member variable. However, the integrator here only uses that value as a starting point: the `Sampler::RoundCount()` method is given an opportunity to change that value to a more appropriate one based on its particular sample generation technique. (For example, many samplers only generate collections of sam-

ples with power-of-two sizes.) The final number of samples for each light is recorded in the `nLightSamples` member variable.

```
(Compute number of samples to use for each light) ≡ 852
    for (const auto &light : scene.lights)
        nLightSamples.push_back(sampler.RoundCount(light->nSamples));

(DirectLightingIntegrator Private Data) +≡ 851
    std::vector<int> nLightSamples;
```

Now the sample arrays can be requested. There are two important details in this fragment: first, although a separate sample request is made for all of the possible ray depths up to the maximum, this doesn't mean that all intersections will have sample arrays available. Rather, once a sample array is retrieved by a call to `Sampler::Get2DArray()`, that array won't be returned again. If both specular reflection and transmission are present, then there may be up to $2^{\maxDepth+1} - 1$ intersection points for each camera ray intersection. If the sample arrays are exhausted, the integrator switches to taking a single sample from each light.

Second, note that the arrays are requested in the order they will be consumed by the integrator: at each intersection point, two arrays are used for each light source, in the same order as the `lights` array.

```
(Request samples for sampling all lights) ≡ 852
    for (int i = 0; i < maxDepth; ++i) {
        for (size_t j = 0; j < scene.lights.size(); ++j) {
            sampler.Request2DArray(nLightSamples[j]);
            sampler.Request2DArray(nLightSamples[j]);
        }
    }

DirectLightingIntegrator 851
DirectLightingIntegrator::
    Li() 853
    DirectLightingIntegrator::
    maxDepth
    852
    DirectLightingIntegrator::
    nLightSamples
    853
    DirectLightingIntegrator::
    strategy
    852
    Light::nSamples 715
    LightStrategy::
    UniformSampleAll
    852
    Sampler::Get2DArray() 424
    Sampler::Request2DArray() 423
    Sampler::RoundCount() 424
    SamplerIntegrator 25
    Scene::lights 23
    UniformSampleAllLights() 854
    UniformSampleOneLight() 856
    WhittedIntegrator::Li() 33
```

As a `SamplerIntegrator`, the main method that the `DirectLightingIntegrator` must implement is `Li()`. The general form of its implementation here is similar to that of `WhittedIntegrator::Li()`: the BSDF at the intersection point is computed, emitted radiance is added if the surface is emissive, rays are traced recursively for specular reflection and transmission, 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 for DirectLightingIntegrator integrator)*, which estimates the value of the integral that gives the reflected radiance, accumulating it into a value `L` that will be returned from `Li()`.

Two helper functions, which other integrators will also find useful, take care of the two sampling strategies.

```
(Compute direct lighting for DirectLightingIntegrator integrator) ≡
    if (strategy == LightStrategy::UniformSampleAll)
        L += UniformSampleAllLights(isect, scene, arena, sampler,
                                     nLightSamples);
    else
        L += UniformSampleOneLight(isect, scene, arena, sampler);
```

To understand the approaches implemented by the two strategies, first recall the scattering equation from Section 5.6, which says that exitant radiance $L_o(p, \omega_o)$ from a point p on a surface in direction ω_o due to incident radiance at the point is given by an 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 will denote by $L_d(p, \omega)$:

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

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

$$\sum_{j=1}^n \int_{S^2} f(p, \omega_o, \omega_i) L_{d(j)}(p, \omega_i) |\cos \theta_i| d\omega_i, \quad [14.12]$$

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

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

One valid approach is to estimate each term of the sum in Equation (14.12) individually, adding the results together. This is the most basic direct lighting strategy and is implemented in `UniformSampleAllLights()`.

In addition to information about the intersection point and additional parameters that are needed to compute the direct lighting, `UniformSampleAllLights()` also takes a `handleMedia` parameter that indicates whether the effects of volumetric attenuation should be considered in the direct lighting computation. (Between this parameter and the detail that it takes an `Interaction` rather than a `SurfaceInteraction`, this function is actually able to compute reflected radiance at points in participating media; fragments related to this functionality will be defined in the next chapter.)

(Integrator Utility Functions) ≡

```
Spectrum UniformSampleAllLights(const Interaction &it,
    const Scene &scene, MemoryArena &arena, Sampler &sampler,
    const std::vector<int> &nLightSamples, bool handleMedia) {
    Spectrum L(0.f);
    for (size_t j = 0; j < scene.lights.size(); ++j) {
        (Accumulate contribution of jth light to L 855)
    }
    return L;
}
```

`Interaction` 115

`MemoryArena` 1074

`Sampler` 421

`Scene` 23

`Scene::lights` 23

`Spectrum` 315

`SurfaceInteraction` 116

`UniformSampleAllLights()` 854

This function attempts to retrieve the sample arrays from the `Sampler` that were previously requested in the `Preprocess()` method.

(Accumulate contribution of jth light to L) ≡

```

const std::shared_ptr<Light> &light = scene.lights[j];
int nSamples = nLightSamples[j];
const Point2f *uLightArray = sampler.Get2DArray(nSamples);
const Point2f *uScatteringArray = sampler.Get2DArray(nSamples);
if (!uLightArray || !uScatteringArray) {
    (Use a single sample for illumination from light 855)
} else {
    (Estimate direct lighting using sample arrays 855)
}

```

854

If all of the requested arrays have been consumed, the code falls back to a single sample estimate via calls to `Sampler::Get2D()`.

(Use a single sample for illumination from light) ≡

```

Point2f uLight = sampler.Get2D();
Point2f uScattering = sampler.Get2D();
L += EstimateDirect(it, uScattering, *light, uLight, scene, sampler,
                    arena, handleMedia);

```

855

For each light sample, the `EstimateDirect()` function (which will be defined shortly) computes the value of the Monte Carlo estimator for its contribution. When sample arrays were successfully obtained, all that remains to be done is to average the estimates from each of their sample values.

(Estimate direct lighting using sample arrays) ≡

```

Spectrum Ld(0.f);
for (int k = 0; k < nSamples; ++k)
    Ld += EstimateDirect(it, uScatteringArray[k], *light, uLightArray[k],
                          scene, sampler, arena, handleMedia);
L += Ld / nSamples;

```

855

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. The Monte Carlo approach 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 2, then the expected value of the result will still be $f(x) + g(x)$. This idea also generalizes to sums with an arbitrary number of terms; 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.

`EstimateDirect()` 858
`Light` 714
`Point2f` 68
`Sampler::Get2D()` 422
`Sampler::Get2DArray()` 424
`Scene::lights` 23
`Spectrum` 315

```
(Integrator Utility Functions) +≡
Spectrum UniformSampleOneLight(const Interaction &it,
    const Scene &scene, MemoryArena &arena, Sampler &sampler,
    bool handleMedia) {
(Randomly choose a single light to sample, light 856)
Point2f uLight = sampler.Get2D();
Point2f uScattering = sampler.Get2D();
return (Float)nLights *
    EstimateDirect(it, uScattering, *light, uLight, scene, sampler,
        arena, handleMedia);
}
```

Which of the `nLights` to sample illumination from is determined using a 1D sample from `Sampler::Get1D()`.

(Randomly choose a single light to sample, light) ≡ 856

```
int nLights = int(scene.lights.size());
if (nLights == 0) return Spectrum(0.f);
int lightNum = std::min((int)(sampler.Get1D() * nLights), nLights - 1);
const std::shared_ptr<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, as 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 these probabilities to reflect the relative contributions of lights to reflected radiance at the reference point, 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 to this task 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 this approach; indeed, any number of samples can be taken in the end, as long as they are weighted appropriately.

14.3.1 ESTIMATING THE DIRECT LIGHTING INTEGRAL

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

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

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

$$\frac{1}{N} \sum_{j=1}^N \frac{f(\mathbf{p}, \omega_o, \omega_j) L_d(\mathbf{p}, \omega_j) |\cos \theta_j|}{p(\omega_j)}.$$

[EstimateDirect\(\)](#) 858
[Float](#) 1062
[Interaction](#) 115
[Light](#) 714
[MemoryArena](#) 1074
[Point2f](#) 68
[Sampler](#) 421
[Sampler::Get1D\(\)](#) 422
[Sampler::Get2D\(\)](#) 422
[Scene](#) 23
[Scene::lights](#) 23
[Spectrum](#) 315
[UniformSampleOneLight\(\)](#) 856

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 14.8 at the end of this chapter for approaches that sample 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 14.13 shows cases where one of the sampling methods is much better than the other. In this scene, four rectangular surfaces ranging from very smooth (top) to

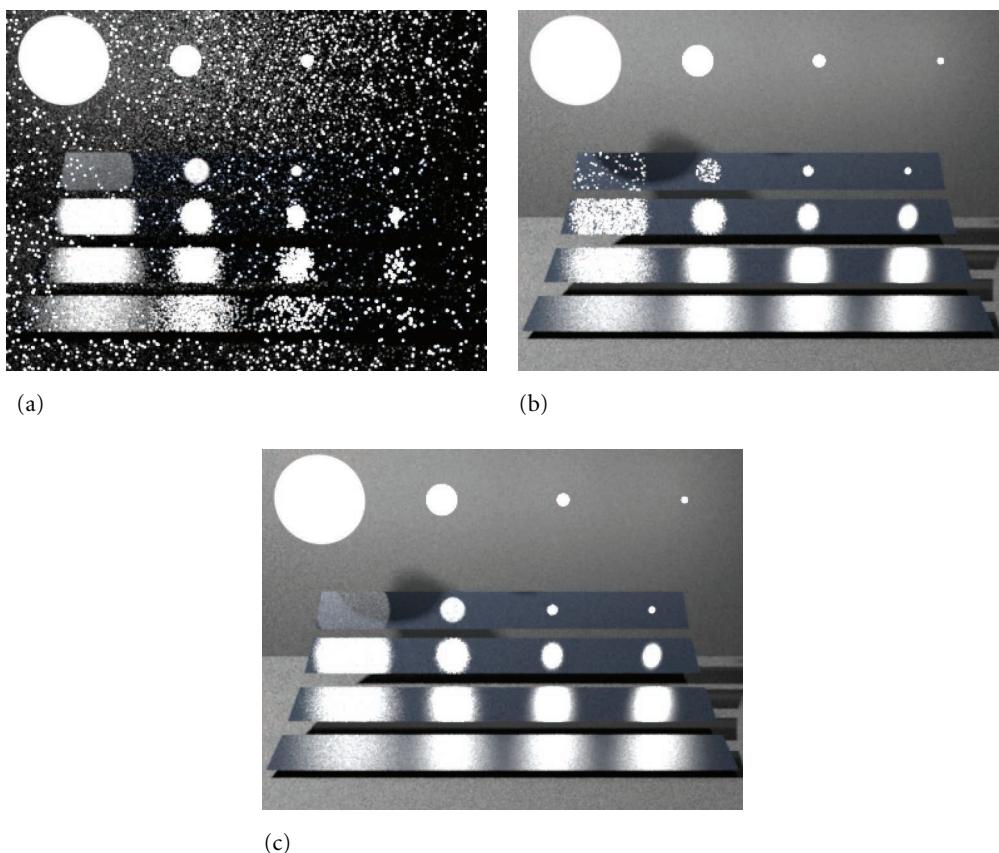


Figure 14.13: Four surfaces ranging from very smooth (top) to very rough (bottom) illuminated by spherical light sources of decreasing size and rendered with different sampling techniques (modeled after a scene by Eric Veach). (a) BSDF sampling, (b) Light sampling, and (c) both techniques combined using MIS. Sampling the BSDF is generally more effective for highly specular materials and large light sources, as illumination is coming from many directions, but the BSDF’s value is large for only a few of them (top left reflection). The converse is true for small sources and rough materials (bottom right reflection), where sampling the light source is more effective.

very rough (bottom) are illuminated by spherical light sources of increasing size. Figures 14.13(a) and (b) show the BSDF and light sampling strategies on their own, while Figure 14.13(c) shows their combination computed using multiple importance sampling. As the example illustrates, sampling the BSDF can be much more effective when it takes on large values on a narrow set of directions that is much smaller than the set of directions that would be obtained by sampling the light sources. This case is most visible in the top left reflection of a large light source in a low-roughness surface. On the other hand, sampling the light sources can be considerably more effective in the opposite case—when the light source is small and the BSDF lobe is less concentrated (this case is most visible in the bottom right reflection).

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 the cases where a sampling method unexpectedly finds a high-contribution direction, since the weighting terms from MIS reduce these contributions substantially.

`EstimateDirect()` implements this approach and computes a direct lighting estimate for a single light source sample. Its `handleMedia` parameter indicates whether the effect of attenuation from participating media should be accounted for, and its `specular` parameter indicates whether or not perfectly specular lobes should be considered in the direct illumination estimate. The default value for both `specular` and `handleMedia` arguments is set to `false` in the function declaration, which is not shown here.

(Integrator Utility Functions) +≡

```
Spectrum EstimateDirect(const Interaction &it,
    const Point2f &uScattering, const Light &light,
    const Point2f &uLight, const Scene &scene, Sampler &sampler,
    MemoryArena &arena, bool handleMedia, bool specular) {
    BxDFType bsdfFlags = specular ? BSDF_ALL :
        BxDFType(BSDF_ALL & ~BSDF_SPECULAR);
    Spectrum Ld(0.f);
    (Sample light source with multiple importance sampling 858)
    (Sample BSDF with multiple importance sampling 860)
    return Ld;
}
```

First, one sample is taken from the light’s sampling distribution using `Sample_Li()`, which also returns the light’s emitted radiance and the value of the PDF for the sampled direction.

(Sample light source with multiple importance sampling) ≡

858

```
Vector3f wi;
Float lightPdf = 0, scatteringPdf = 0;
VisibilityTester visibility;
Spectrum Li = light.Sample_Li(it, uLight, &wi, &lightPdf, &visibility);
```

BSDF_ALL 513
 BSDF_SPECULAR 513
 BxDFType 513
 Float 1062
 Interaction 115
 Light 714
 Light::Sample_Li() 716
 MemoryArena 1074
 Point2f 68
 Sampler 421
 Scene 23
 Spectrum 315
 Spectrum::IsBlack() 317
 Vector3f 60
 VisibilityTester 717

```

    if (lightPdf > 0 && !Li.IsBlack()) {
        <Compute BSDF or phase function's value for light sample 859>
        if (!f.IsBlack()) {
            <Compute effect of visibility for light source sample 859>
            <Add light's contribution to reflected radiance 860>
        }
    }
}

```

Only if the light successfully samples a direction and returns nonzero emitted radiance does `EstimateDirect()` go ahead and evaluate the BSDF or phase function at the provided `Interaction`; otherwise, there's no reason to go through the computational expense. (Consider, for example, a spotlight, which returns no radiance for points outside its illumination cone.)

```

<Compute BSDF or phase function's value for light sample> ≡
Spectrum f;
if (it.IsSurfaceInteraction()) {
    <Evaluate BSDF for light sampling strategy 859>
} else {
    <Evaluate phase function for light sampling strategy 900>
}

```

858

```

<Evaluate BSDF for light sampling strategy> ≡
const SurfaceInteraction &isect = (const SurfaceInteraction &)it;
f = isect.bsdf->f(isect.wo, wi, bsdfFlags) * AbsDot(wi, isect.shading.n);
scatteringPdf = isect.bsdf->Pdf(isect.wo, wi, bsdfFlags);

```

859

(We postpone the medium-specific fragment that evaluates the phase function at the interaction point, `<Evaluate medium reflectance for light sampling strategy>`, to Chapter 15.)

If participating media are to be accounted for, the 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. Otherwise, a call to the `VisibilityTester`'s `Unoccluded()` method traces a shadow ray to determine if the sampled point on the light source is visible. (This step and the next are skipped if the BSDF or phase function returned a black SPD.)

```

<Compute effect of visibility for light source sample> ≡
if (handleMedia)
    Li *= visibility.Tr(scene, sampler);
else if (!visibility.Unoccluded(scene))
    Li = Spectrum(0.f);

```

858

AbsDot() 64
BSDF::f() 575
BSDF::Pdf() 834
Interaction::
 IsSurfaceInteraction()
 116
Spectrum 315
SurfaceInteraction 116
SurfaceInteraction::bsdf 250
SurfaceInteraction::shading
 118
SurfaceInteraction::
 shading::n
 118
VisibilityTester 717
VisibilityTester::Tr() 718
VisibilityTester::
 Unoccluded()
 718

The light sample's contribution can now be accumulated. Recall from Section 14.2.1 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_Li()` 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's PDF value

for sampling the direction ω_i , which was returned by `BSDF::Pdf()`, is used with the MIS estimator, where the weight is computed here with the power heuristic.

```
(Add light's contribution to reflected radiance) ≡ 858
if (!Li.IsBlack()) {
    if (IsDeltaLight(light.flags))
        Ld += f * Li / lightPdf;
    else {
        Float weight = PowerHeuristic(1, lightPdf, 1, scatteringPdf);
        Ld += f * Li * weight / lightPdf;
    }
}
```

Next, a sample is generated using the BSDF's sampling distribution. This step should be skipped if the light source's emission profile involves 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. Otherwise, the BSDF can be sampled.

```
(Sample BSDF with multiple importance sampling) ≡ 858
if (!IsDeltaLight(light.flags)) {
    Spectrum f;
    bool sampledSpecular = false;
    if (it.IsSurfaceInteraction()) {
        (Sample scattered direction for surface interactions 860)
    } else {
        (Sample scattered direction for medium interactions 900)
    }
    if (!f.IsBlack() && scatteringPdf > 0) {
        (Account for light contributions along sampled direction wi 861)
    }
}
```

Once more, we postpone the medium-related code to Chapter 15. Given a surface interaction, the implementation samples a scattered direction and records whether or not a delta distribution was sampled.

```
(Sample scattered direction for surface interactions) ≡ 860
BxDFType sampledType;
const SurfaceInteraction &isect = (const SurfaceInteraction &)it;
f = isect.bsdf->Sample_f(isect.wo, &wi, uScattering, &scatteringPdf,
                           bsdfFlags, &sampledType);
f *= AbsDot(wi, isect.shading.n);
sampledSpecular = sampledType & BSDF_SPECULAR;
```

AbsDot() 64
`BSDF::Pdf()` 834
`BSDF::Sample_f()` 832
`BSDF_SPECULAR` 513
`BxDFType` 513
`Float` 1062
`Interaction::IsSurfaceInteraction()` 116
`IsDeltaLight()` 715
`Light::flags` 715
`PowerHeuristic()` 799
`Spectrum` 315
`Spectrum::IsBlack()` 317
`SurfaceInteraction` 116

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.

```

⟨Account for light contributions along sampled direction wi⟩ ≡ 860
    Float weight = 1;
    if (!sampledSpecular) {
        lightPdf = light.Pdf_Li(it, wi);
        if (lightPdf == 0)
            return Ld;
        weight = PowerHeuristic(1, scatteringPdf, 1, lightPdf);
    }
    ⟨Find intersection and compute transmittance 861⟩
    ⟨Add light contribution from material sampling 861⟩

```

Given a direction sampled by the BSDF or a medium's phase function, 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. When participating media are being accounted for, the transmittance up to the intersection point on the light is recorded.

```

⟨Find intersection and compute transmittance⟩ ≡ 861
    SurfaceInteraction lightIsect;
    Ray ray = it.SpawnRay(wi);
    Spectrum Tr(1.f);
    bool foundSurfaceInteraction = handleMedia ?
        scene.IntersectTr(ray, sampler, &lightIsect, &Tr) :
        scene.Intersect(ray, &lightIsect);

```

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 material sampling⟩ ≡ 861
    Spectrum Li(0.f);
    if (foundSurfaceInteraction) {
        if (lightIsect.primitive->GetAreaLight() == &light)
            Li = lightIsect.Le(-wi);
    }
    else
        Li = light.Le(ray);
    if (!Li.IsBlack())
        Ld += f * Li * Tr * weight / scatteringPdf;

```

Float 1062
GeometricPrimitive::
GetAreaLight()
251
InfiniteAreaLight 737
Interaction::SpawnRay() 232
Light::Le() 741
Light::Pdf_Li() 836
PowerHeuristic() 799
Ray 73
Scene::Intersect() 24
Scene::IntersectTr() 687
Spectrum 315
Spectrum::IsBlack() 317
SurfaceInteraction 116
SurfaceInteraction::Le() 734
SurfaceInteraction::primitive
249

14.4 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. For now we will continue only to consider

the case where there are no participating media in the scene. (Chapter 15 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 that will be implemented in Chapter 16.

14.4.1 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.

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:

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

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 L_o 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 14.14. 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' :

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

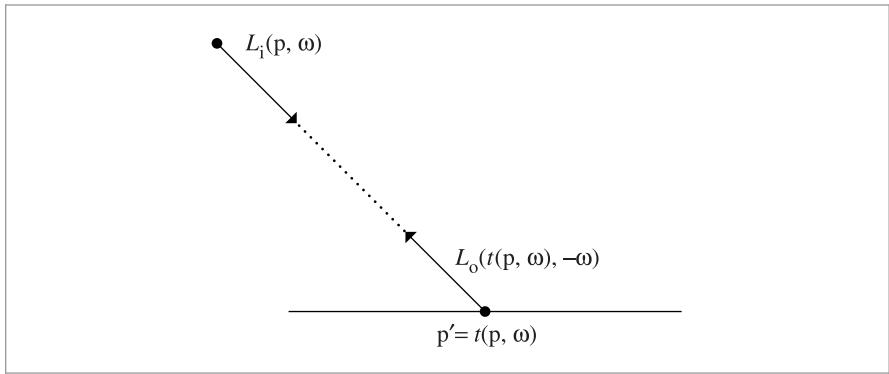


Figure 14.14: Radiance along a Ray through Free Space Is Unchanged. Therefore, to compute the incident radiance along a ray from point p in direction ω , we can find the first surface the ray intersects and compute exitant radiance in the direction $-\omega$ there. The trace operator $t(p, \omega)$ gives the point p' on the first surface that the ray (p, ω) intersects.

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 0.

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

$$L(p, \omega_o) = L_e(p, \omega_o) + \int_{S^2} f(p, \omega_o, \omega_i) L(t(p, \omega_i), -\omega_i) |\cos \theta_i| d\omega_i. \quad (14.13)$$

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.

14.4.2 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 among 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, $f(p, \omega_o, \omega_i) = c$, and also emit a constant amount of radiance in all directions. We have

$$L(p, \omega_o) = L_e + c \int_{\mathcal{H}^2(n)} L(t(p, \omega_i), -\omega_i) |\cos \theta_i| d\omega_i.$$

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

$$L = L_e + c\pi L.$$

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

$$\begin{aligned} L &= L_e + \rho_{hh}(L_e + \rho_{hh}(L_e + \dots \\ &= \sum_{i=0}^{\infty} L_e \rho_{hh}^i. \end{aligned}$$

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

$$L = \frac{L_e}{1 - \rho_{hh}}.$$

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.⁷ For example, the `DirectLightingIntegrator` integrator effectively computes the result of making a single substitution:

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

where

$$L_d = L_e(t(p, \omega_i), -\omega_i)$$

and further scattering is ignored.

⁷ 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 1995a) and Veach's thesis (Veach 1997) for further information.

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.

14.4.3 THE SURFACE FORM OF THE LTE

One reason why the LTE as written in Equation (14.13) is complex is that the relationship between geometric objects in the scene is implicit in the ray-tracing function $t(p, \omega)$. Making the behavior of this function explicit in the integrand will shed some light on the structure of this equation. To do this, we will rewrite Equation (14.13) 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

$$L(p' \rightarrow p) = L(p', \omega)$$

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

$$f(p'' \rightarrow p' \rightarrow p) = f(p', \omega_o, \omega_i),$$

where $\omega_i = \widehat{p'' - p'}$ and $\omega_o = \widehat{p - p'}$ (Figure 14.15).

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 $|\cos \theta'|/r^2$.

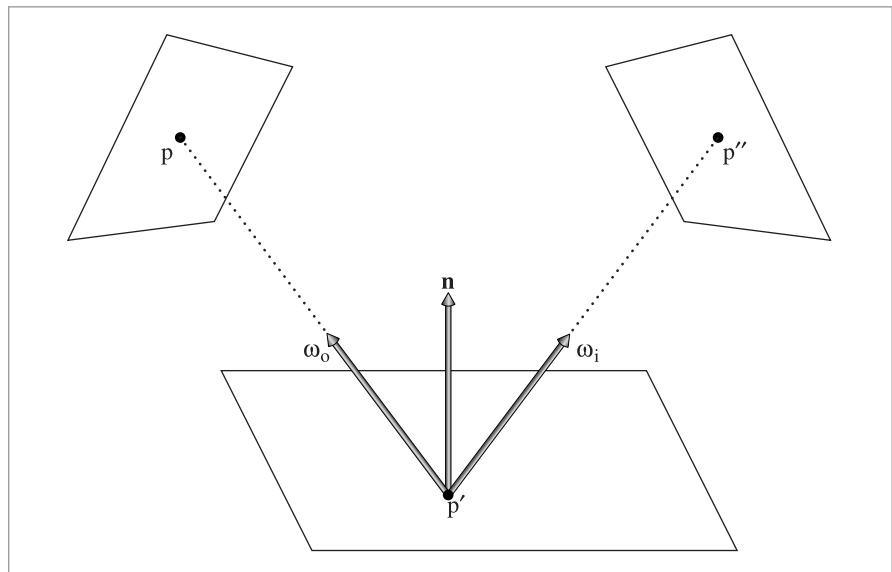


Figure 14.15: The three-point form of the light transport equation converts the integral to be over the domain of points on surfaces in the scene, rather than over directions over the sphere. It is a key transformation for deriving the path integral form of the light transport equation.

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, $G(p \leftrightarrow p')$:

$$G(p \leftrightarrow p') = V(p \leftrightarrow p') \frac{|\cos \theta| |\cos \theta'|}{\|p - p'\|^2}. \quad [14.14]$$

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

$$L(p' \rightarrow p) = L_e(p' \rightarrow p) + \int_A f(p'' \rightarrow p' \rightarrow p) L(p'' \rightarrow p') G(p'' \leftrightarrow p') dA(p''), \quad [14.15]$$

where A is all of the surfaces of the scene.

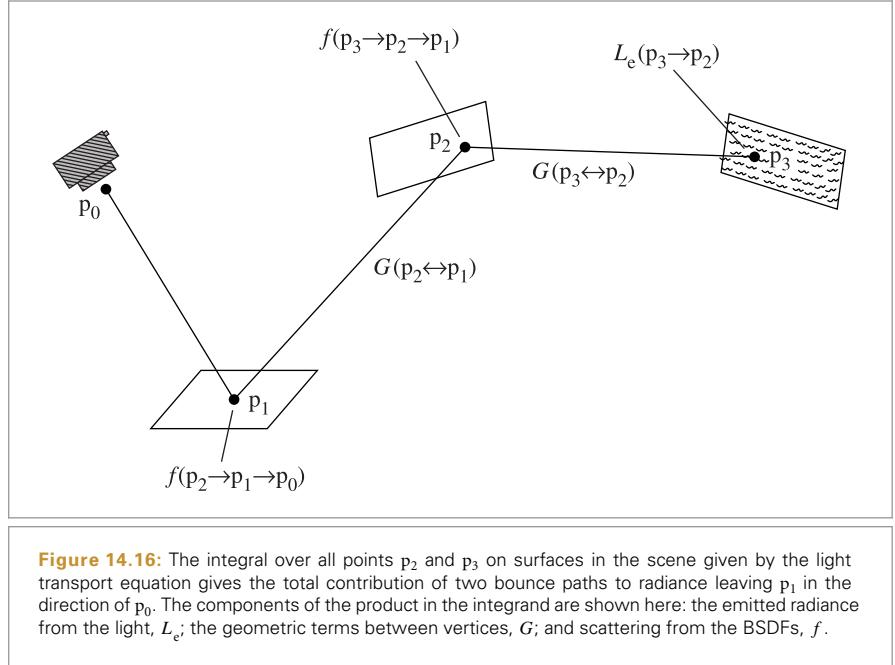
Although Equations (14.13) and (14.15) are equivalent, they represent two different ways of approaching light transport. To evaluate Equation (14.13) with Monte Carlo, we would sample a number of directions from a distribution of directions on the sphere and cast rays to evaluate the integrand. For Equation (14.15), 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 $V(p \leftrightarrow p')$.

14.4.4 INTEGRAL OVER PATHS

With the area integral form of Equation (14.15), we can derive a more flexible form of the LTE known as the *path integral* formulation of light transport, which expresses radiance as an integral over paths that are themselves points in a high dimensional *path space*. One of the main motivations for using path space is that it provides an expression for the value of a measurement as an explicit integral over paths, as opposed to the unwieldy recursive definition resulting from the energy balance equation, (14.13).

The explicit form allows for considerable freedom in how these paths are found—essentially any technique for randomly choosing paths can be turned into a workable rendering algorithm that computes the right answer given a sufficient number of samples. This form of the LTE provides the foundation of the bidirectional light transport algorithms in Chapter 16.

To go from the area integral to a sum over path integrals involving 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 $L(p'' \rightarrow p')$ 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 direction $p_1 - p_0$:



$$\begin{aligned}
 L(p_1 \rightarrow p_0) = & L_e(p_1 \rightarrow p_0) \\
 & + \int_A L_e(p_2 \rightarrow p_1) f(p_2 \rightarrow p_1 \rightarrow p_0) G(p_2 \leftrightarrow p_1) dA(p_2) \\
 & + \int_A \int_A L_e(p_3 \rightarrow p_2) f(p_3 \rightarrow p_2 \rightarrow p_1) G(p_3 \leftrightarrow p_2) \\
 & \quad \times f(p_2 \rightarrow p_1 \rightarrow p_0) G(p_2 \leftrightarrow p_1) dA(p_3) dA(p_2) + \dots
 \end{aligned}$$

Each term on the right side of this equation represents a path of increasing length. For example, the third term is illustrated in Figure 14.16. 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 origin 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

$$L(p_1 \rightarrow p_0) = \sum_{n=1}^{\infty} P(\bar{p}_n). \quad [14.16]$$

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

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

where p_0 is on the film plane or front lens element and p_n is on a light source, and

$$P(\bar{p}_n) = \underbrace{\int_A \int_A \cdots \int_A}_{n-1} L_e(p_n \rightarrow p_{n-1}) \\ \times \left(\prod_{i=1}^{n-1} f(p_{i+1} \rightarrow p_i \rightarrow p_{i-1}) G(p_{i+1} \leftrightarrow p_i) \right) dA(p_2) \cdots dA(p_n). \quad [14.17]$$

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

$$T(\bar{p}_n) = \prod_{i=1}^{n-1} f(p_{i+1} \rightarrow p_i \rightarrow p_{i-1}) G(p_{i+1} \leftrightarrow p_i), \quad [14.18]$$

so

$$P(\bar{p}_n) = \underbrace{\int_A \int_A \cdots \int_A}_{n-1} L_e(p_n \rightarrow p_{n-1}) T(\bar{p}_n) dA(p_2) \cdots dA(p_n).$$

Given Equation (14.16) 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 $P(\bar{p}_n)$ 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 to practical light transport algorithms throughout this and the following two chapters.

14.4.5 DELTA DISTRIBUTIONS IN THE INTEGRAND

Delta functions may be present in $P(\bar{p}_i)$ 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). If present, these distributions need to be handled explicitly by the light transport algorithm. 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, $P(\bar{p}_2)$, in a scene with a single point light source at point p_{light} described by a delta distribution:

$$\begin{aligned} P(\bar{p}_2) &= \int_A L_e(p_2 \rightarrow p_1) f(p_2 \rightarrow p_1 \rightarrow p_0) G(p_2 \leftrightarrow p_1) dA(p_2) \\ &= \frac{\delta(p_{\text{light}} - p_2) L_e(p_{\text{light}} \rightarrow p_1)}{p(p_{\text{light}})} f(p_2 \rightarrow p_1 \rightarrow p_0) G(p_2 \leftrightarrow p_1). \end{aligned}$$

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 $p(p_{\text{light}})$ (recall the discussion of sampling delta distributions in Section 14.1.3), 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 $T(\bar{p}_n)$; each one eliminates an integral over area from the estimate to be computed.

14.4.6 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. Section 16.2.2 will introduce the concept of density estimation, which is used to implement a rendering algorithm known as *stochastic progressive photon mapping* (SPPM). The underlying density estimation that algorithm uses works well on diffuse surfaces because scattered radiance only depends on the surface position in this case to store a 2D radiance discretization, but for glossy surfaces it becomes preferable to switch to other techniques such as path tracing.

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

$$L(p_1 \rightarrow p_0) = P(\bar{p}_1) + P(\bar{p}_2) + \sum_{i=3}^{\infty} P(\bar{p}_i),$$

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 $P(\bar{p}_1)$ and $P(\bar{p}_2)$ with the algorithm that handles $P(\bar{p}_3)$ and beyond (and similarly with the other terms).

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

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

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 $L_{e,s}$ integral ignores any emission from large lights, the estimate of the $L_{e,l}$ 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 $f_{-\Delta}$ denotes the remaining components,

$$\begin{aligned}
 P(\bar{\mathbf{p}}_n) &= \int_{A^{n-1}} L_e(\mathbf{p}_n \rightarrow \mathbf{p}_{n-1}) \\
 &\quad \times \prod_{i=1}^{n-1} (f_\Delta(\mathbf{p}_{i+1} \rightarrow \mathbf{p}_i \rightarrow \mathbf{p}_{i-1}) + f_{-\Delta}(\mathbf{p}_{i+1} \rightarrow \mathbf{p}_i \rightarrow \mathbf{p}_{i-1})) \\
 &\quad \times G(\mathbf{p}_{i+1} \leftrightarrow \mathbf{p}_i) dA(\mathbf{p}_2) \cdots dA(\mathbf{p}_n).
 \end{aligned}$$

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 $f_{-\Delta}$ components; all of the terms like $f_\Delta f_{-\Delta} f_{-\Delta}$ must be accounted for as well if a partitioning scheme like this is used.

14.5 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 14.17 compares images of a scene rendered with 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



(a)



(b)

Figure 14.17: 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. (*Model courtesy of Guillermo M. Leal Llaguno.*)

understanding of the path integral equation and will make the generalization to bidirectional path tracing (Section 16.3) easier to understand.

14.5.1 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 ,

$$L(p_1 \rightarrow p_0) = \sum_{i=1}^{\infty} P(\bar{p}_i),$$

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 $P(\bar{p}_i)$ terms with a finite amount of computation?
2. Given a particular $P(\bar{p}_i)$ term, how do we generate one or more paths \bar{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 $P(\bar{p}_i)$ and will then start to apply Russian roulette to stop sampling after a finite number of terms without introducing bias. Recall from Section 13.7 that Russian roulette allows us to probabilistically stop computing terms in a sum as long as we reweight the terms that are not terminated. For example, if we always computed estimates of $P(\bar{p}_1)$, $P(\bar{p}_2)$, and $P(\bar{p}_3)$ but stopped without computing more terms with probability q , then an unbiased estimate of the sum would be

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

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 ,

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

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 $P(\bar{p}_i)$ and because it will be weighted appropriately if we do evaluate it, the final result is an unbiased estimate of the sum.

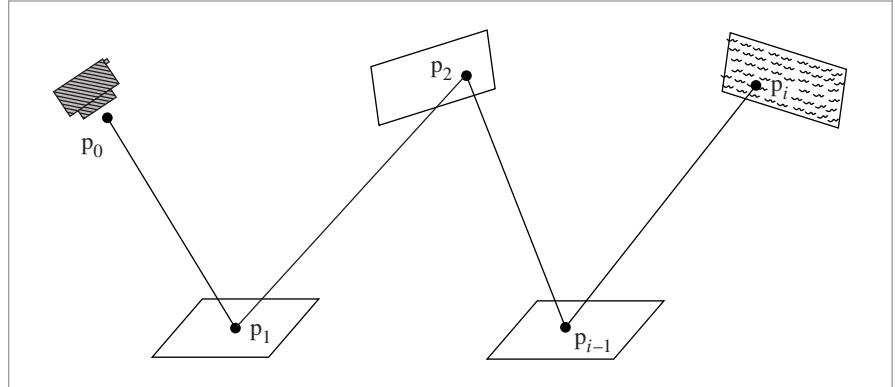


Figure 14.18: A path \bar{p}_i of $i + 1$ vertices from the camera at p , intersecting a series of positions on surfaces in the scene, to a point on the light p_i . Scattering according to the BSDF occurs at each path vertex from p_i to p_{i-1} such that the radiance estimate at the camera due to this path is given by the product of the path throughput $T(\bar{p}_i)$ and the emitted radiance from the light divided by the path sampling weights.

14.5.2 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 $P(\bar{p}_i)$. 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 a point on the camera film or lens (Figure 14.18). Looking at the form of $P(\bar{p}_i)$, 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 i th object should be

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

Then, given a method to sample a point on the i th 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:

$$p_A(p_i) = \frac{1}{\sum_j A_j}.$$

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 p_0, p_1, \dots, 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 $P(\bar{p}_i)$; 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 high 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. 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 $f(p_{i+1} \rightarrow p_i \rightarrow p_{i-1})$ 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.

14.5.3 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):

$$p_A = p_\omega \frac{|\cos \theta_i|}{\|p_i - p_{i+1}\|^2}.$$

This correction causes all of the terms of the geometric term $G(p_i \leftrightarrow p_{i+1})$ to cancel out of $P(\bar{p}_i)$ except for the $\cos \theta_{i+1}$ term. 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 equal to 1. An alternative way to think about this is that ray tracing provides an operation to importance sample the visibility component of G . 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 $p_A(p_i)$, the value of the Monte Carlo estimate for a path is

$$\begin{aligned} & \frac{L_e(p_i \rightarrow p_{i-1}) f(p_i \rightarrow p_{i-1} \rightarrow p_{i-2}) G(p_i \leftrightarrow p_{i-1})}{p_A(p_i)} \\ & \times \left(\prod_{j=1}^{i-2} \frac{f(p_{j+1} \rightarrow p_j \rightarrow p_{j-1}) |\cos \theta_j|}{p_\omega(p_{j+1} - p_j)} \right). \end{aligned} \quad (14.19)$$

14.5.4 IMPLEMENTATION

Our path-tracing implementation computes an estimate of the sum of path contributions $P(\bar{p}_i)$ 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 $p_A(p_i)$ 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.

Beyond how lights are sampled, another small difference is that as the estimates of the path contribution terms $P(\bar{p}_i)$ 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 $P(\bar{p}_i)$ 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 SamplerIntegrator {
public:
    <PathIntegrator Public Methods 876>
private:
    <PathIntegrator Private Data 876>
};
```

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 value if only Russian roulette should be used to terminate paths.

(PathIntegrator Public Methods) ≡

875

```
PathIntegrator(int maxDepth, std::shared_ptr<const Camera> camera,
              std::shared_ptr<Sampler> sampler)
: SamplerIntegrator(camera, sampler), maxDepth(maxDepth) { }
```

(PathIntegrator Private Data) ≡

875

```
const int maxDepth;
```

A number of variables record the current state of the path. beta holds the *path throughput weight*, which is defined as the factors of the throughput function $T(\bar{p}_{i-1})$ —i.e., the product of the BSDF values and cosine terms for the vertices generated so far, divided by their respective sampling PDFs:

$$\beta = \prod_{j=1}^{i-2} \frac{f(p_{j+1} \rightarrow p_j \rightarrow p_{j-1}) |\cos \theta_j|}{p_\omega(p_{j+1} - p_j)}.$$

Thus, the product of beta with scattered light from direct lighting at the final vertex of the path gives the contribution for a path. (This quantity will reoccur many times in the following two chapters, and we will consistently refer to it as beta.) Because the effect of earlier path vertices is aggregated in this way, there is no need to store the positions and BSDFs of all of the vertices of the path, only the last one.

In the following implementation, L holds the radiance value from the running total of $\sum P(\bar{p}_i)$, 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.

(PathIntegrator Method Definitions) ≡

```
Spectrum PathIntegrator::Li(const RayDifferential &r, const Scene &scene,
                            Sampler &sampler, MemoryArena &arena, int depth) const {
    Spectrum L(0.f), beta(1.f);
    RayDifferential ray(r);
    bool specularBounce = false;
    for (int bounces = 0; ; ++bounces) {
        (Find next path vertex and accumulate contribution 877)
    }
    return L;
}
```

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.

Camera 356
MemoryArena 1074
PathIntegrator 875
RayDifferential 75
Sampler 421
SamplerIntegrator 25
Scene 23
Spectrum 315

(Find next path vertex and accumulate contribution) ≡ 876
(Intersect ray with scene and store intersection in isect 877)
(Possibly add emitted light at intersection 877)
(Terminate path if ray escaped or maxDepth was reached 877)
(Compute scattering functions and skip over medium boundaries 878)
(Sample illumination from lights to find path contribution 878)
(Sample BSDF to get new path direction 878)
(Account for subsurface scattering, if applicable 915)
(Possibly terminate the path with Russian roulette 879)

The first step in the loop is to find the next path vertex by intersecting ray against the scene geometry.

(Intersect ray with scene and store intersection in isect) ≡ 877, 900
 SurfaceInteraction isect;
 bool foundIntersection = scene.Intersect(ray, &isect);

If the ray hits an object that is emissive, the emission is usually ignored, since the loop iteration at the previous path vertex performed a direct illumination estimate that already accounted for its effect. The same is true when a ray escapes into an emissive environment. However, there are two exceptions: 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 is when the sampled direction from the last path vertex was from a specular BSDF component: in this case, the previous iteration's direct illumination estimate could not evaluate the associated integrand containing a Dirac delta function, and we must account for it here.

(Possibly add emitted light at intersection) ≡ 877
 if (bounces == 0 || specularBounce) {
(Add emitted light at path vertex or from the environment 877)
}

When no intersection is found, the ray has escaped the scene and thus the path sampling iteration terminates. Similarly, the iteration terminates when bounces exceeds the prescribed maximum value.

(Terminate path if ray escaped or maxDepth was reached) ≡ 877
 if (!foundIntersection || bounces >= maxDepth)
 break;

When emitted light should be included, the path throughput weight must be multiplied with the radiance emitted by the current path vertex (if an intersection was found) or radiance emitted by infinite area light sources, if present.

(Add emitted light at path vertex or from the environment) ≡ 877
 if (foundIntersection)
 L += beta * isect.Le(-ray.d);
else
 for (const auto &light : scene.lights)
 L += beta * light->Le(ray);

Light::Le() 741
PathIntegrator::maxDepth 876
Scene::Intersect() 24
Scene::lights 23
SurfaceInteraction 116
SurfaceInteraction::Le() 734

Before estimating the direct illumination at the current vertex, it is necessary to compute the scattering functions at the vertex. A special case arises when `SurfaceInteraction::bsdf` is equal to `nullptr`, which indicates that the current surface has no effect on light. pbrt uses such surfaces to represent transitions between participating media, whose boundaries are themselves optically inactive (i.e., they have the same index of refraction on both sides). Since the basic `PathIntegrator` ignores media, it simply skips over such surfaces without counting them as scattering events in the bounces counter.

(Compute scattering functions and skip over medium boundaries) ≡ 877

```
isect.ComputeScatteringFunctions(ray, arena, true);
if (!isect.bsdf) {
    ray = isect.SpawnRay(ray.d);
    bounces--;
    continue;
}
```

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 path throughput weight gives its overall contribution to the total radiance estimate.

(Sample illumination from lights to find path contribution) ≡ 877

```
L += beta * UniformSampleOneLight(isect, scene, arena, sampler);
```

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 integrator updates the path throughput weight 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) ≡ 877

```
Vector3f wo = -ray.d, wi;
Float pdf;
BxDFType flags;
Spectrum f = isect.bsdf->Sample_f(wo, &wi, sampler.Get2D(),
    &pdf, BSDF_ALL, &flags);
if (f.IsBlack() || pdf == 0.f)
    break;
beta *= f * AbsDot(wi, isect.shading.n) / pdf;
specularBounce = (flags & BSDF_SPECULAR) != 0;
ray = isect.SpawnRay(wi);
```

AbsDot() 64
`BSDF::Sample_f()` 832
`BSDF_ALL` 513
`BSDF_SPECULAR` 513
`BxDFType` 513
`Float` 1062
`Interaction::SpawnRay()` 232
`Sampler::Get2D()` 422
`Spectrum` 315
`Spectrum::IsBlack()` 317
`SurfaceInteraction::bsdf` 250
`SurfaceInteraction::ComputeScatteringFunctions()` 578
`SurfaceInteraction::shading` 118
`SurfaceInteraction::shading::n` 118
`UniformSampleOneLight()` 856
`Vector3f` 60

The case where a ray refracts into a material with a BSSRDF is handled specially in the fragment *(Account for subsurface scattering, if applicable)*, which is implemented in Section 15.4.3 after subsurface scattering has been discussed in more detail.

Path termination kicks in after a few bounces, with termination probability q set based on the path throughput weight. In general, it's worth having a higher probability of terminating low-contributing paths, since they have relatively less impact on the final image. (A minimum termination probability ensures termination is possible if β is

large; for example, due to a large BSDF value divided by a low sampling probability.) If the path isn't terminated, beta is updated with the Russian roulette weight and all subsequent $P(\bar{p}_i)$ terms will be appropriately affected by it.

```
(Possibly terminate the path with Russian roulette) ≡ 877, 900
    if (bounces > 3) {
        Float q = std::max((Float).05, 1 - beta.y());
        if (sampler.Get1D() < q)
            break;
        beta /= 1 - q;
    }
```

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).

Additional important theoretical work on light transport has been done by Arvo (1993, 1995a), 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).

Monte Carlo Techniques

Russian roulette and splitting were 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 more desirable artifact.

Cook and collaborators first introduced random sampling for integration in rendering (Cook, Porter, and Carpenter 1984; Cook 1986), and Kajiya (1986) developed the general-purpose path-tracing algorithm. Other important early work on Monte Carlo in rendering includes Shirley's Ph.D. thesis (1990) and a paper by Kirk and Arvo (1991) on sources of bias in rendering algorithms. Shirley (1992) described a number of useful recipes for warping uniform random numbers to useful distributions for rendering.

Float 1062
Sampler::Get1D() 422
Spectrum::y() 325

Keller and collaborators have written extensively on the application of quasi-Monte Carlo integration in graphics (Keller 1996, 2001; Friedel and Keller 2000; Kollig and Keller 2000, 2002). Keller's "Quasi-Monte Carlo image synthesis in a nutshell" (2012) is a good introduction to quasi-Monte Carlo for rendering.

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

Many researchers have investigated techniques for adaptive sampling, adding more samples in parts of the integrand that are complex, and adaptive reconstruction, where final results are reconstructed from noisy samples with more sophisticated techniques than the simple reconstruction filters from Chapter 7. See Zwicker et al.’s survey article (2015) for a thorough summary of recent work in this area.

Sampling BSDFs

The approach for directly sampling the microfacet visible normal distribution implemented in Section 14.1.1 was developed by Heitz and d’Eon (2014). See also Heitz (2014a) for an overview of traditional sampling techniques for various microfacet distribution functions that sample the regular D function directly.

When dealing with refraction through rough dielectrics, a modified change of variables term is needed to account for the mapping from half vectors to outgoing direction. A model based on this approach was originally developed by Stam (2001); Walter et al. (2007) proposed improvements and provided an alternative geometric justification of the half vector mapping.

Lawrence et al. (2004) developed methods for sampling arbitrary BRDF models, including those based on measured reflectance data. They applied methods that factor the 4D BRDF into a product of two 2D functions, both of which are guaranteed to always be greater than 0, thus making it possible to use them as importance sampling distributions. Most of the recently developed parametric models for fitting BRDFs described in Chapter 8’s “Further Reading” section have been developed with importance sampling in mind. The sampling technique for the FourierBSDF was developed by Jakob et al. (2014a).

A number of researchers have investigated effective sampling of hair reflection models (typically Marschner et al.’s model (2003)). See, for example, papers by Ou et al. (2012), Hery and Ramamoorthi (2012), and d’Eon et al. (2013). More recently, Pekelis et al. (2015) developed a more efficient approach to sampling the Marschner model.

The idea of applying statistical hypothesis tests to verify the correctness of graphics-related Monte Carlo sampling routines such as BSDF models was introduced by Subr and Arvo (2007a). The χ^2 test variant for validating the BSDF model implementations in pbrt was originally developed as part of the Mitsuba renderer by Jakob (2010).

Direct Lighting

Algorithms to render soft shadows from area lights were first developed by Amanatides (1984) and Cook, Porter, and Carpenter (1984). Shirley et al. (1996) derived methods for sampling a number of shapes for use as area light sources. Arvo showed how to sample the projection of a triangle on the sphere of directions with respect to a reference point; this approach can give better results than sampling the area of the triangle directly (Arvo 1995b). Ureña et al. (2013) and Pekelis and Hery (2014) developed analogous techniques for sampling projected quadrilateral light sources. The approach implemented in Sec-

tion 14.2.2 to convert an angle (θ, ϕ) in a cone to a point on a sphere was derived by Akalin (2015).

Subr and Arvo (2007b) developed an efficient technique for sampling environment map light sources that not only accounts for the $\cos \theta$ term from the scattering equation but also only generates samples in the hemisphere around the surface normal.

When environment maps are used for illuminating indoor scenes, many incident directions may be blocked by the building structure. Bashford-Rogers et al. (2013) developed a two-pass algorithm where a first pass from the camera finds directions that reach the environment map; this information is used to create sampling distributions that are used during a second rendering pass. Bitterli et al. (2015) developed an interesting solution to this issue: they rectify the environment map so that rectangular portals in the building map to rectangular regions of the environment map. In turn, at a given point receiving illumination, they compute the projection of portals to the outside and can efficiently sample the environment map using a summed area table.

As described in the “Further Reading” section of Chapter 12, a useful generalization of environment maps for illumination allows the emitted radiance to vary by both position and direction. Lu et al. (2015) developed techniques for efficiently importance sampling these light sources and describes previous work in this area.

The expense of tracing shadow rays to light sources can be significant; a number of interesting approaches have been developed to improve the efficiency of this part of the rendering computation. Billen et al. (2013) propose a technique where only a random subset of potential occluders are tested for intersections; a compensation term ensures that the result is unbiased. Following work shows how to use simplified geometry for some shadow tests while still computing the correct result overall (Billen et al. 2014). Another approach to reducing the cost of shadow rays is visibility caching, where the point-to-point visibility function’s value is cached for clusters of points on surfaces in the scene (Clarberg and Akenine-Möller 2008b; Popov et al. 2013).

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 (1991) 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” algorithm, described in the “Further Reading” section of Chapter 16, also addresses this problem.

`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 from either one individually. A number of more efficient approaches have been developed to sample directly from the

product distribution (Burke 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; Clarberg and Akenine-Möller 2008a). 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.

Subr et al. (2014) analyzed the combination of multiple importance sampling and jittered sampling for direct lighting calculations and propose sampling improvements to improve convergence rates.

Other Topics

With full spectral rendering, it's sometimes necessary to perform a sampling operation for a ray based on a single wavelength from the SPD (e.g., with wavelength-dependent indices of refraction). For this case, Radziszewski et al. (2009) introduced an application of multiple importance sampling that reduces variance in this case.

Ward and collaborators developed the irradiance caching algorithm, which is described in a series of papers (Ward, Rubinstein, and Clear 1988; Ward 1994). The basic idea is to cache irradiance from indirect illumination at a sparse set of points on surfaces in the scene; because indirect lighting is generally slowly changing, irradiance can often be safely interpolated. Tabellion and Lamorlette (2004) described a number of additional improvements to irradiance caching that made it viable for rendering for movie productions. 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. 2005). Recent work by Schwarzhaft et al. proposed a better way of assessing the validity of a cache point using a second-order expansion of the incident lighting (Schwarzhaft et al. 2012).

EXERCISES

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

- ➋ 14.2 Fix the buggy `Sphere::Sample()` and `Disk::Sample()` methods, which currently don't properly account for partial spheres and disks when they sample points

`BSDF::Pdf()` 834

`BSDF::Sample_f()` 832

`BxDF` 513

`Disk::Sample()` 838

`Material` 577

`Sphere::Sample()` 840

on the surface. Create a scene that demonstrates the error from the current implementations and for which your solution is clearly an improvement.

- ② 14.3 It is possible to derive a sampling method for cylinder area light sources that only chooses points over the visible area as seen from the receiving point, similar to the improved sphere sampling method in this chapter (Gardner et al. 1987; Zimmerman 1995). Learn more about these methods, or rederive them yourself, and write a new implementation of `Cylinder::Sample()` that implements such an algorithm. Verify that pbrt still generates correct images with your method, and measure how much the improved version reduces variance for a fixed number of samples taken. How much does it improve efficiency? How do you explain any discrepancy between the amount of reduction in variance and the amount of improvement in efficiency?
- ③ 14.4 The sampling approach implemented in `InfiniteAreaLight::Sample_Li()` is ineffective in scenes like indoor environments, where many directions are occluded by the building structure. One approach to this problem is to manually provide a representation of *portals* like windows that the light passes through. These portals can then be sampled by area to find paths that lead to the light. However, such an approach doesn't account for the directional radiance distribution of the light source.
Bitterli et al. (2015) suggested an improved method that causes rectangular portals to be rectangular areas in the environment map, which in turn can be sampled directly. Implement their approach as a new `InfiniteAreaLight` sampling method in pbrt, and measure the improvement in efficiency for rendering scenes where there is a substantial amount of occlusion between the part of the scene being rendered and the infinite light source.
- ④ 14.5 The infinite area light importance sampling method implemented in this chapter doesn't perfectly match the distribution of the light source's emission distribution: recall that the emission function is computed with bilinear interpolation among image samples, but the sampling distribution is computed as a piecewise-constant function of a slightly blurred version of the texture map. In some cases, this discrepancy can lead to high variance when there are localized extremely bright texels. (In the worst case, consider a source that has a very small value at all texels except one, which is 10,000 times brighter than all of the others.) In that case, the value of the function may be much higher than predicted by the sampling PDFs at points very close to that sample, thus leading to high variance ($f(x)/p(x)$ is large).

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

`Cylinder::Sample()` 839

`InfiniteAreaLight` 737

`InfiniteAreaLight::
 Sample_Li()`
849

lighting calculations but to use bilinear filtering for camera rays and rays that have only undergone specular reflection?

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

- ➊ 14.6 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 implement this approach, 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.
- ➋ 14.7 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.
- ➌ 14.8 Implement a technique for generating samples from the product of the light and BSDF distributions; see the papers by Burke 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.
- ➍ 14.9 Clarberg and Akenine-Möller (2008b) and Popov et al. (2013) both described algorithms that performs visibility caching, computing, and interpolating information about light source visibility at points in the scene. Implement one of these methods 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?
- ➎ 14.10 Investigate algorithms for rendering scenes with large numbers of light sources: see, for example, the papers by Ward (1991a), Shirley, Wang, and Zimmerman (1996), 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.
- ➏ 14.11 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.

CHAPTER FIFTEEN

15 LIGHT TRANSPORT II: VOLUME RENDERING

Just as BSDFs characterize reflection from the surfaces in a scene, Medium class implementations represent scattering that occurs *between* surfaces; examples include atmospheric scattering effects such as haze, absorption in a stained glass window, or scattering by fat globules in a bottle of milk. Technically, all of these phenomena are due to surface interactions with a vast number of microscopic particles, though it is preferable to find a less cumbersome way of modeling them than considering them individually. With the models described in this chapter, the particles are assumed to be so numerous that they can be represented using statistical distributions instead of an explicit enumeration.

This chapter begins with an introduction of the equation of transfer, which describes the equilibrium distribution of radiance in scenes with participating media, and then presents a number of sampling methods that are useful for Monte Carlo integration with participating media. Given this foundation, the `VolPathIntegrator` can be introduced—it extends the `PathIntegrator` to solve the light transport equation in the presence of participating media.

After Section 15.4 describes how to sample from BSSRDF distributions, Section 15.5 then describes the implementation of a BSSRDF that models the aggregate light scattering in media bounded by refractive surfaces. Although the approach is expressed in terms of radiance leaving from and arriving at surfaces, it is included in this chapter since its implementation is based on an approximate solution to the equation of transfer in participating media.

15.1 THE EQUATION OF TRANSFER

The equation of transfer is the fundamental equation that governs the behavior of light in a medium that absorbs, emits, and scatters radiation. It accounts for all of the volume scattering processes described in Chapter 11—absorption, emission, and in- and out-scattering—to give an equation that describes the distribution of radiance in an environment. The light transport equation is in fact a special case of the equation of transfer, simplified by the lack of participating media and specialized for scattering from surfaces.

In its most basic form, the equation of transfer is an integro-differential equation that describes how the radiance along a beam changes at a point in space. It can be transformed into a pure integral equation that describes the effect of participating media from the infinite number of points along a ray. It can be derived in a straightforward manner by subtracting the effects of the scattering processes that reduce energy along a beam (absorption and out-scattering) from the processes that increase energy along it (emission and in-scattering).

Recall the source term L_s from Section 11.1.4: it gives the change in radiance at a point p in a particular direction ω due to emission and in-scattered light from other points in the medium:

$$L_s(p, \omega) = L_e(p, \omega) + \sigma_s(p, \omega) \int_{S^2} p(p, \omega', \omega) L_i(p, \omega') d\omega'.$$

The source term accounts for all of the processes that add radiance to a ray.

The attenuation coefficient, $\sigma_t(p, \omega)$, accounts for all processes that reduce radiance at a point: absorption and out-scattering. The differential equation that describes its effect is

$$dL_o(p + t\omega, \omega) = -\sigma_t(p, \omega) L_i(p, -\omega) dt.$$

The overall differential change in radiance at a point p' along a ray is found by adding these two effects together to get the integro-differential form of the equation of transfer:¹

$$\frac{\partial}{\partial t} L_o(p + t\omega, \omega) = -\sigma_t(p, \omega) L_i(p, -\omega) + L_s(p, \omega). \quad (15.1)$$

With suitable boundary conditions, this equation can be transformed to a pure integral equation. For example, if we assume that there are no surfaces in the scene so that the rays are never blocked and have an infinite length, the integral equation of transfer is

$$L_i(p, \omega) = \int_0^\infty T_r(p' \rightarrow p) L_s(p', -\omega) dt,$$

where $p' = p + t\omega$ (Figure 15.1). The meaning of this equation is reasonably intuitive: it just says that the radiance arriving at a point from a given direction is contributed to by the added radiance along all points along the ray from the point. The amount of added radiance at each point along the ray that reaches the ray's origin is reduced by the total beam transmittance from the ray's origin to the point.

¹ It is an integro-differential equation due to the integral over the sphere in the source term.

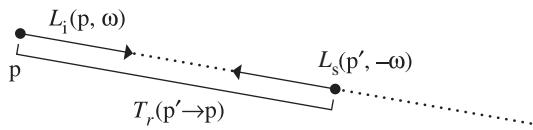


Figure 15.1: The equation of transfer gives the incident radiance at point $L_i(p, \omega)$ accounting for the effect of participating media. At each point along the ray, the source term $L_s(p', -\omega)$ gives the differential radiance added at the point due to scattering and emission. This radiance is then attenuated by the beam transmittance $T_r(p' \rightarrow p)$ from the point p' to the ray's origin.

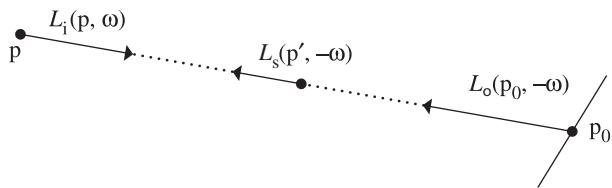


Figure 15.2: For a finite ray that intersects a surface, the incident radiance, $L_i(p, \omega)$, is equal to the outgoing radiance from the surface, $L_o(p_0, -\omega)$, times the beam transmittance to the surface plus the added radiance from all points along the ray from p to p_0 .

More generally, if there are reflecting and/or emitting surfaces in the scene, rays don't necessarily have infinite length and the first surface that a ray hits affects its radiance, adding outgoing radiance from the surface at the point and preventing radiance from points along the ray beyond the intersection point from contributing to radiance at the ray's origin. If a ray (p, ω) intersects a surface at some point p_0 at a parametric distance t along the ray, then the integral equation of transfer is

$$L_i(p, \omega) = T_r(p_0 \rightarrow p)L_o(p_0, -\omega) + \int_0^t T_r(p' \rightarrow p)L_s(p', -\omega)dt', \quad [15.2]$$

where $p_0 = p + t\omega$ is the point on the surface and $p' = p + t'\omega$ are points along the ray (Figure 15.2).

This equation describes the two effects that contribute to radiance along the ray. First, reflected radiance back along the ray from the surface is given by the L_o term, which gives the emitted and reflected radiance from the surface. This radiance may be attenuated by the participating media; the beam transmittance from the ray origin to the point p_0 accounts for this. The second term accounts for the added radiance along the ray due to volume scattering and emission but only up to the point where the ray intersects the surface; points beyond that one don't affect the radiance along the ray.

* 15.1.1 GENERALIZED PATH SPACE

Just as it was helpful to express the LTE as a sum over paths of scattering events, it's also helpful to express the integral equation of transfer in this form. Doing so is a prerequisite for constructing participating medium-aware bidirectional integrators in Chapter 16.

Recall how in Section 14.4.4, the surface form of the LTE was repeatedly substituted into itself to derive the path space contribution function for a path of length n

$$P(\bar{p}_n) = \underbrace{\int_A \int_A \cdots \int_A}_{n-1} L_e(p_n \rightarrow p_{n-1}) T(\bar{p}_n) dA(p_2) \cdots dA(p_n),$$

where the throughput $T(\bar{p}_n)$ was defined as

$$T(\bar{p}_n) = \prod_{i=1}^{n-1} f(p_{i+1} \rightarrow p_i \rightarrow p_{i-1}) G(p_{i+1} \leftrightarrow p_i).$$

This previous definition only works for surfaces, but using a similar approach of substituting the integral equation of transfer, a medium-aware path integral can be derived. The derivation is laborious and we will just present the final result here. Refer to Pauly et al. (2000) and Chapter 3 of Jakob's Ph.D. thesis (2013) for a full derivation.

Previously, integration occurred over a Cartesian product of surface locations A^n . Now, we'll need a formal way of writing down an integral that can consider an arbitrary sequence of both 2D surface locations A and 3D positions in a participating medium V . First, we'll focus only on a specific arrangement of n surface and medium vertices encoded in a binary configuration vector c . The associated set of paths is given by a Cartesian product of surface locations and medium locations,

$$\mathcal{P}_n^c = \prod_{i=1}^n \begin{cases} A, & \text{if } c_i = 0 \\ V, & \text{if } c_i = 1. \end{cases}$$

The set of all paths of length n is the union of the above sets over all possible configuration vectors:

$$\mathcal{P}_n = \bigcup_{c \in \{0,1\}^n} \mathcal{P}_n^c.$$

Next, we define a *measure*, which provides an abstract notion of the volume of a subset $D \subseteq \mathcal{P}_n$ that is essential for integration. The measure we'll use simply sums up the product of surface area and volume associated with the individual vertices in each of the path spaces of specific configurations.

$$\mu_n(D) = \sum_{c \in \{0,1\}^n} \mu_n^c(D \cap \mathcal{P}_n^c) \quad \text{where } \mu_n^c(D) = \int_D \prod_{i=1}^n \begin{cases} dA(p_i), & \text{if } c_i = 0 \\ dV(p_i), & \text{if } c_i = 1. \end{cases}$$

The generalized path contribution $\hat{P}(\bar{p}_n)$ can now be written as

$$\hat{P}(\bar{p}_n) = \int_{\mathcal{P}_{n-1}} L_e(p_n \rightarrow p_{n-1}) \hat{T}(\bar{p}_n) d\mu_{n-1}(p_2, \dots, p_n).$$

Due to the measure defined earlier, this is really a sum of many integrals considering all possible sequences of surface and volume scattering events.

In this framework, the path throughput function $\hat{T}(\bar{p}_n)$ is defined as:

$$\hat{T}(\bar{p}_n) = \prod_{i=1}^{n-1} \hat{f}(p_{i+1} \rightarrow p_i \rightarrow p_{i-1}) \hat{G}(p_{i+1} \leftrightarrow p_i). \quad [15.3]$$

It now refers to a generalized scattering distribution function \hat{f} and geometric term \hat{G} . The former simply falls back to the BSDF or phase function (multiplied by σ_s) depending on the type of the vertex p_i .

$$\hat{f}(p_{i+1} \rightarrow p_i \rightarrow p_{i-1}) = \begin{cases} \sigma_s p(p_{i+1} \rightarrow p_i \rightarrow p_{i-1}), & \text{if } p_i \in V \\ f(p_{i+1} \rightarrow p_i \rightarrow p_{i-1}), & \text{if } p_i \in A. \end{cases} \quad [15.4]$$

Equation (14.14) in Section 14.4.3 originally defined the geometric term G as

$$G(p \leftrightarrow p') = V(p \leftrightarrow p') \frac{|\cos \theta| |\cos \theta'|}{\|p - p'\|^2}.$$

A generalized form of this geometric term is given by

$$\hat{G}(p \leftrightarrow p') = V(p \leftrightarrow p') T_r(p \rightarrow p') \frac{C_p(p, p') C_{p'}(p', p)}{\|p - p'\|^2}, \quad [15.5]$$

where the T_r term now also accounts for transmittance between the two points, and

$$C_p(p, p') = \begin{cases} \left| \mathbf{n}_p \cdot \frac{p - p'}{\|p - p'\|} \right|, & \text{if } p \text{ is a surface vertex} \\ 1, & \text{otherwise} \end{cases}$$

only incorporates the absolute angle cosine between the connection segment and the normal direction when the underlying vertex p is located on a surface.

15.2 SAMPLING VOLUME SCATTERING

Before proceeding to algorithms that model the effect of light scattering in participating media, we'll first define some building-block functionality for sampling from distributions related to participating media and for computing the beam transmittance for spatially varying media.

The `Medium` interface defines a `Sample()` method, which takes a world space ray (p, ω) and possibly samples a medium scattering interaction along it. The input ray will generally have been intersected against the scene geometry; thus, implementations of this method shouldn't ever sample a medium interaction at a point on the ray beyond its t_{\max} value. Without loss of generality, the following discussion assumes that there is always a surface at some distance $t_{\max} < \infty$.

`MediumInteraction` 688

`MemoryArena` 1074

`Ray` 73

`Sampler` 421

`Spectrum` 315

$\langle Medium\ Interface \rangle + \equiv$

```
virtual Spectrum Sample(const Ray &ray, Sampler &sampler,
    MemoryArena &arena, MediumInteraction *mi) const = 0;
```

684

The objective of this method is to sample the integral form of the equation of transfer, Equation (15.2), which consists of a surface and a medium-related term:

$$L_i(p, \omega) = T_r(p_0 \rightarrow p)L_o(p_0, -\omega) + \int_0^t T_r(p + t\omega \rightarrow p)L_s(p + t\omega, -\omega) dt,$$

where $p_0 = p + t_{\max}\omega$ is the point on the surface. We will neglect the effect of medium emission and assume directionally constant medium properties, in which case the source term is given by

$$L_s(p, \omega) = \sigma_s(p) \int_{S^2} p(p, \omega', \omega) L_i(p, \omega') d\omega'. \quad [15.6]$$

Two cases can occur: if `Sample()` doesn't sample an interaction on the given ray interval $[0, t_{\max}]$, then the surface-related term $T_r(p_0 \rightarrow p)L_o(p_0, -\omega)$ should be estimated. If it does sample an interaction, the second integral term is to be estimated, and the provided `MediumInteraction` should be initialized accordingly.

Suppose that $p_t(t)$ denotes the probability per unit distance of generating an interaction at position $p + t\omega$. Due to the possibility of not sampling a medium interaction, this function generally doesn't integrate to 1, and we define p_{surf} as the associated discrete probability of sampling the surface term:

$$p_{\text{surf}} = 1 - \int_0^{t_{\max}} p_t(t) dt$$

With these definitions, we can now specify the semantics of `Sample()`, which differs from previously encountered techniques for scattering functions like `BSDF::Sample_f()` in that it does not provide the caller with separate information about the function value and PDF at the sampled position. This information is not generally needed, and some medium models (specifically, the heterogeneous medium) admit more efficient sampling schemes when it is possible to compute *ratios* of these quantities instead.

When the surface term is selected, the method should return a weight equal to

$$\beta_{\text{surf}} = \frac{T_r(p \rightarrow p + t\omega)}{p_{\text{surf}}}, \quad [15.7]$$

which corresponds to sampling the first summand. Note that the value of the outgoing radiance $L_o(p_0, -\omega)$ is not included in β_{surf} ; it is the responsibility of the caller to account for this term. In the medium case, the method returns

$$\beta_{\text{med}} = \frac{\sigma_s(p + t\omega) T_r(p \rightarrow p + t\omega)}{p_t(t)}, \quad [15.8]$$

which corresponds to sampling all medium-related terms except for the integral over in-scattered light in Equation (15.6), which must be handled separately.

The scattering coefficient and transmittance allow for spectral variation, hence this method returns a `Spectrum`-valued weighting factor to update the path throughput weight β up to the surface or medium scattering event.

As is generally the case for Monte Carlo integration, estimators like β_{surf} and β_{med} admit a variety of sampling techniques that all produce the desired distribution. The implementation of the heterogeneous medium will make use of this fact to provide an implementation that is considerably more efficient than the canonical sampling approach based on the inversion method.

So that calling code can easily determine whether the provided `MediumInteraction` was initialized by `Sample()`, `MediumInteraction` provides an `IsValid()` method that takes

`BSDF::Sample_f()` 832

`MediumInteraction` 688

advantage of the fact that any time a medium scattering event has been sampled, the phase function pointer will be set.

```
<MediumInteraction Public Methods> +≡
    bool IsValid() const { return phase != nullptr; }
```

688

15.2.1 HOMOGENEOUS MEDIUM

The `HomogeneousMedium` implementation of this method is fairly straightforward; the only complexities come from needing to handle attenuation coefficients that vary by wavelength.

```
<HomogeneousMedium Method Definitions> +≡
    Spectrum HomogeneousMedium::Sample(const Ray &ray, Sampler &sampler,
                                         MemoryArena &arena, MediumInteraction *mi) const {
        <Sample a channel and distance along the ray 894>
        <Compute the transmittance and sampling density 894>
        <Return weighting factor for scattering from homogeneous medium 894>
    }
```

In Section 13.3.1 we derived the sampling method for an exponential distribution defined over $[0, \infty)$. For $f(t) = e^{-\sigma_t t}$, it is

$$t = -\frac{\ln(1 - \xi)}{\sigma_t}, \quad (15.9)$$

with PDF

$$p_t(t) = \sigma_t e^{-\sigma_t t}. \quad (15.10)$$

However, the attenuation coefficient σ_t in general varies by wavelength. It is not desirable to sample multiple points in the medium, so a uniform sample is first used to select a spectral channel i ; the corresponding scalar σ_t^i value is then used to sample a distance along the distribution

$$\hat{p}_t^i(t) = \sigma_t^i e^{-\sigma_t^i t},$$

using the technique from Equation (15.9). The resulting sampling density is the average of the individual strategies p_t^i :

$$\hat{p}_t(t) = \frac{1}{n} \sum_{i=1}^n \sigma_t^i e^{-\sigma_t^i t}. \quad (15.11)$$

The (discrete) probability of sampling a surface interaction at $t = t_{\max}$ is the complement of generating a medium scattering event between $t = 0$ and $t = t_{\max}$. This works out to a probability equal to the average transmittance over all n spectral channels:

$$p_{\text{surf}} = 1 - \int_0^{t_{\max}} \hat{p}_t(t) dt = \frac{1}{n} \sum_{i=1}^n e^{-\sigma_t^i t_{\max}}. \quad (15.12)$$

MediumInteraction 688
 MediumInteraction::phase 688
 MemoryArena 1074
 Ray 73
 Sampler 421
 Spectrum 315

The implementation draws a sample according to Equation (15.11); if the sampled distance is before the ray-primitive intersection (if any), then a medium scattering event is recorded by initializing the `MediumInteraction`. Otherwise, the sampled point in the medium is ignored, and corresponding surface interaction should be used as the next

path vertex by the integrator. This sampling approach is naturally efficient: the probability of generating a medium interaction instead of the surface interaction is exactly equal to 1 minus the beam transmittance for the selected wavelength. Thus, given optically thin media (or a short ray extent), the surface interaction is more often used, and for thick media (or longer rays), a medium interaction is more likely to be sampled.

(Sample a channel and distance along the ray) ≡ 893

```
int channel = std::min((int)(sampler.Get1D() * Spectrum::nSamples),
                      Spectrum::nSamples - 1);
Float dist = -std::log(1 - sampler.Get1D()) / sigma_t[channel];
Float t = std::min(dist * ray.d.Length(), ray.tMax);
bool sampledMedium = t < ray.tMax;
if (sampledMedium)
    *mi = MediumInteraction(ray(t), -ray.d, ray.time, this,
                           ARENA_ALLOC(arena, HenyeyGreenstein)(g));
```

In either case, the beam transmittance Tr is easily computed using Beer's law, Equation (11.3), just as in the `HomogeneousMedium::Tr()` method.

(Compute the transmittance and sampling density) ≡ 893

```
Spectrum Tr = Exp(-sigma_t * std::min(t, MaxFloat) * ray.d.Length());
```

Finally, the method computes the sample density using Equations (15.11) or (15.12) and returns resulting sampling weight β_{surf} and β_{med} , depending on the value of `sampledMedium`.

(Return weighting factor for scattering from homogeneous medium) ≡ 893

```
Spectrum density = sampledMedium ? (sigma_t * Tr) : Tr;
Float pdf = 0;
for (int i = 0; i < Spectrum::nSamples; ++i)
    pdf += density[i];
pdf *= 1 / (Float)Spectrum::nSamples;
return sampledMedium ? (Tr * sigma_s / pdf) : (Tr / pdf);
```

15.2.2 HETEROGENEOUS MEDIUM

In the case of the `GridDensityMedium`, extra effort is necessary to deal with the medium's heterogeneous nature. When the spatial variation can be decomposed into uniform regions (e.g., piecewise constant voxels), a technique known as *regular tracking* applies standard homogeneous medium techniques to the voxels individually; a disadvantage of this approach is that it becomes costly when there are many voxels. Since the `GridDensityMedium` relies on linear interpolation, this approach cannot be used.

Other techniques build on a straightforward generalization of the homogeneous sampling PDF from Equation (15.10) with a spatially varying attenuation coefficient:

$$p_t(t) = \sigma_t(t) e^{- \int_0^t \sigma_t(t') dt'}, \quad [15.13]$$

where $\sigma_t(t) = \sigma_t(p + t\omega)$ evaluates the attenuation at distance t along the ray. The most commonly used method for importance sampling Equation (15.13), is known as *ray*

ARENA_ALLOC() [576](#)
CoefficientSpectrum::nSamples [318](#)
Float [1062](#)
GridDensityMedium [690](#)
HenyeyGreenstein [682](#)
HomogeneousMedium::g [688](#)
HomogeneousMedium::sigma_t [689](#)
HomogeneousMedium::Tr() [689](#)
MaxFloat [210](#)
MediumInteraction [688](#)
Ray::time [73](#)
Ray::tMax [73](#)
Sampler::Get1D() [422](#)
Spectrum [315](#)
Spectrum::Exp() [317](#)
Vector3::Length() [65](#)

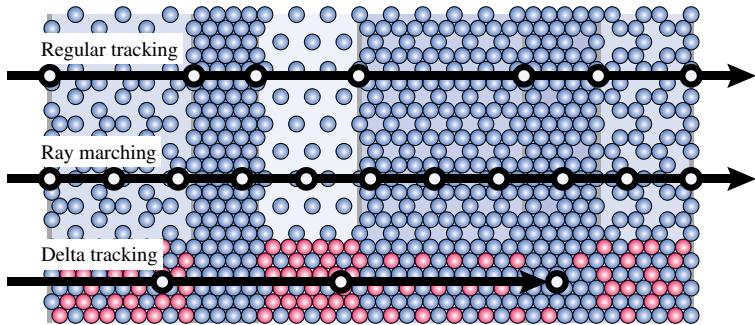


Figure 15.3: Ray Marching and Delta Tracking in a Medium with Density that Varies Along the Horizontal Axis. (top) Regular tracking partitions the medium into a number of homogeneous sub-regions and relies on standard techniques for dealing with the individual homogeneous regions. (middle) Ray marching partitions the ray into a number of discrete segments and approximates the transmittance through each one. (bottom) Delta tracking effectively considers a medium that is “filled” with additional virtual particles (red) until it reaches a uniform density. Image courtesy of Novák et al. (2014).

marching. This method inverts an approximate cumulative distribution by partitioning the range $[0, t_{\max}]$ into a number of subintervals, numerically approximating the integral in each interval, and finally inverting this discrete representation. Unfortunately discretizing the problem in this way introduces systemic statistical bias, which means that an Integrator using ray marching generally won’t converge to the right result (even when an infinite number of samples per pixel is used). Furthermore, this bias can manifest itself in the form of distracting visual artifacts.

For this reason, we prefer an alternative unbiased approach proposed by Woodcock et al. (1965) that was originally developed to simulate volumetric scattering of neutrons in atomic reactors. This technique is known as *delta tracking* and is easiest to realize when the attenuation coefficient σ_t is monochromatic. Our implementation includes an assertion test (not shown here) to verify that this is indeed the case. Note that the scattering and absorption coefficients are still permitted to vary with respect to wavelength—however, their sum $\sigma_t = \sigma_s + \sigma_a$ must be uniform.

Figure 15.3 compares regular tracking, ray marching, and delta tracking. Delta tracking can be interpreted as filling the medium with additional (virtual) particles until its attenuation coefficient is constant everywhere. Sampling the resulting homogeneous medium is then easily accomplished using the basic exponential scheme from Equation (15.9). However, whenever an interaction with a particle occurs, it is still necessary to determine if it involved a “real” or a “virtual” particle (in which case the interaction is disregarded). The elegant insight of Woodcock et al. was that this decision can be made randomly based on the local fraction of “real” particles, which leads to a distribution of samples matching Equation (15.13).

```
GridDensityMedium::  
GridDensityMedium()  
690
```

Integrator 25

The following fragment is part of the `GridDensityMedium::GridDensityMedium()` constructor; its purpose is to precompute the inverse of the maximum density scale factor over the entire medium, which will be a useful quantity in the delta tracking implementation discussed next.

(Precompute values for Monte Carlo sampling of GridDensityMedium) ≡ **690**

```
sigma_t = (sigma_a + sigma_s)[0];
Float maxDensity = 0;
for (int i = 0; i < nx * ny * nz; ++i)
    maxDensity = std::max(maxDensity, density[i]);
invMaxDensity = 1 / maxDensity;
```

(GridDensityMedium Private Data) +≡ **690**

```
Float sigma_t;
Float invMaxDensity;
```

The `Sample()` method begins by transforming the ray into the medium coordinate system and normalizing the ray direction; `ray.tMax` is scaled appropriately to account for the normalization.

(GridDensityMedium Method Definitions) +≡

```
Spectrum GridDensityMedium::Sample(const Ray &rWorld, Sampler &sampler,
    MemoryArena &arena, MediumInteraction *mi) const {
    Ray ray = WorldToMedium(Ray(rWorld.o, Normalize(rWorld.d),
        rWorld.tMax * rWorld.d.Length()));
    (Compute [tmin, tmax] interval of ray's overlap with medium bounds 896)
    (Run delta-tracking iterations to sample a medium interaction 897)
}
```

Next, the implementation computes the parametric range of the ray's overlap with the medium's bounds, which are the unit cube $[0, 1]^3$. This step is technically not required for correct operation but is generally a good idea: reducing the length of the considered ray segment translates into a correspondingly smaller number of delta tracking iterations.

(Compute [t_{min}, t_{max}] interval of ray's overlap with medium bounds) ≡ **896, 898**

```
const Bounds3f b(Point3f(0, 0, 0), Point3f(1, 1, 1));
Float tMin, tMax;
if (!b.IntersectP(ray, &tMin, &tMax))
    return Spectrum(1.f);
```

Assuming that the maximum extinction value throughout the medium is given by $\sigma_{t,\max}$, each delta-tracking iteration i performs a standard exponential step through the uniform medium:

$$t_i = t_{i-1} - \frac{\ln(1 - \xi_{2i})}{\sigma_{t,\max}},$$

where $t_0 = t_{\min}$. These steps are repeated until one of two stopping criteria is satisfied: first, if $t_i > t_{\max}$ then we have left the medium without an interaction and `Medium::Sample()` hasn't sampled a scattering event. Alternatively, the loop may be terminated at each iteration i with probability $\sigma_t(t_i)/\sigma_{t,\max}$, the local fraction of "real" particles. This random decision consumes ξ_{2i+1} , the second of two uniform samples per iteration i .

```
Bounds3::IntersectP() 127
Bounds3f 76
Float 1062
GridDensityMedium::
    invMaxDensity
    896
GridDensityMedium::sigma_t
    896
GridDensityMedium::
    WorldToMedium
    690
Medium::Sample() 891
MediumInteraction 688
MemoryArena 1074
Point3f 68
Ray 73
Ray::tMax 73
Sampler 421
Spectrum 315
Vector3::Length() 65
Vector3::Normalize() 66
```

```
<Run delta-tracking iterations to sample a medium interaction> ≡ 896
    Float t = tMin;
    while (true) {
        t -= std::log(1 - sampler.Get1D()) * invMaxDensity / sigma_t;
        if (t >= tMax)
            break;
        if (Density(ray(t)) * invMaxDensity > sampler.Get1D()) {
            (Populate mi with medium interaction information and return 897)
        }
    }
    return Spectrum(1.f);
```

The probability of not sampling a medium interaction is equal to the transmittance of the ray segment $[t_{\min}, t_{\max}]$; hence 1.0 is returned for the sampling weight β_{surf} according to Equation (15.7). The medium interaction case resembles the fragment *(Sample a channel and distance along the ray)*.

```
(Populate mi with medium interaction information and return) ≡ 897
    PhaseFunction *phase = ARENA_ALLOC(arena, HenyeyGreenstein)(g);
    *mi = MediumInteraction(rWorld(t), -rWorld.d, rWorld.time, this, phase);
    return sigma_s / sigma_t;
```

Finally, we must also provide an implementation of the `Tr()` method to compute the transmittance along a ray segment. Consider the pseudocode of the following simplistic implementation that performs a call to `Sample()` and returns 1.0 if the ray passed through the segment $[0, t_{\max}]$ and 0.0 when a medium interaction occurred along the way. This effectively turns the transmittance function into a binary random variable.

```
ARENA_ALLOC() 576
Float 1062
GridDensityMedium::Density() 691
GridDensityMedium::g 690
GridDensityMedium::invMaxDensity 896
GridDensityMedium::sigma_s 690
GridDensityMedium::sigma_t 896
GridDensityMedium::Tr() 898
HenyeyGreenstein 682
MediumInteraction 688
PhaseFunction 681
Sampler::Get1D() 422
Spectrum 315
```

Since the probability of passing through the medium is equal to the transmittance, this random variable has the correct mean and could be used in the context of unbiased Monte Carlo integration. Calling `Tr()` many times and averaging the result would produce an increasingly accurate estimate of the transmittance, though this will generally be too costly to do in practice. On the other hand, using the naive binary implementation leads to a high amount of variance.

Novák et al. (2014) observed that this binary-valued function can be interpreted as an instance of Russian roulette. However, instead of randomly terminating the algorithm with a value of zero in each iteration, we could also remove the Russian roulette logic and simply multiply the transmittance by the probability of continuation. The resulting estimator has the same mean with a considerably lower variance. We will use this approach in the implementation of `GridDensityMedium::Tr()`.

```
(GridDensityMedium Method Definitions) +≡
Spectrum GridDensityMedium::Tr(const Ray &rWorld,
                                  Sampler &sampler) const {
    Ray ray = WorldToMedium(Ray(rWorld.o, Normalize(rWorld.d),
                                rWorld.tMax * rWorld.d.Length()));
    (Compute [tmin, tmax] interval of ray's overlap with medium bounds 896)
    (Perform ratio tracking to estimate the transmittance value 898)
}
```

The beginning of the `Tr()` method matches `Sample()`. The loop body is also identical except for the last line, which multiplies a running product by the ratio of real particles to hypothetical particles. (Novák referred to this scheme as *ratio tracking*).

(Perform ratio tracking to estimate the transmittance value) ≡ 898

```
Float Tr = 1, t = tMin;
while (true) {
    t -= std::log(1 - sampler.Get1D()) * invMaxDensity / sigma_t;
    if (t >= tMax)
        break;
    Float density = Density(ray(t));
    Tr *= 1 - std::max((Float)0, density * invMaxDensity);
}
return Spectrum(Tr);
```

15.2.3 SAMPLING PHASE FUNCTIONS

It is also useful to be able to draw samples from the distribution described by phase functions—applications include applying multiple importance sampling to computing direct lighting in participating media as well as for sampling scattered directions for indirect lighting samples in participating media. For these applications, `PhaseFunction` implementations must implement the `Sample_p()` method, which samples an incident direction ω_i given the outgoing direction ω_o and a sample value in $[0, 1]^2$.

Note that, unlike the `BxDF` sampling methods, `Sample_p()` doesn't return both the phase function's value and its PDF. Rather, `pbrt` assumes that phase functions are sampled with PDFs that perfectly match their distributions. In conjunction with the requirement that phase functions themselves be normalized (Equation (11.4)), a single return value encodes both values. When the value of the PDF alone is needed, a call to `PhaseFunction::p()` suffices.

(PhaseFunction Interface) +≡ 681

```
virtual Float Sample_p(const Vector3f &wo, Vector3f *wi,
                       const Point2f &u) const = 0;
```

The PDF for the Henyey–Greenstein phase function is separable into θ and ϕ components, with $p(\phi) = 1/(2\pi)$ as usual. The main task is to sample $\cos \theta$.

`BxDF` 513
`Float` 1062
`GridDensityMedium::Density()` 691
`GridDensityMedium::invMaxDensity` 896
`GridDensityMedium::sigma_t` 896
`GridDensityMedium::WorldToMedium` 690
`PhaseFunction::p()` 681
`Point2f` 68
`Ray` 73
`Ray::tMax` 73
`Sampler` 421
`Sampler::Get1D()` 422
`Spectrum` 315
`Vector3f::Length()` 65
`Vector3f::Normalize()` 66
`Vector3f` 60

```

⟨HenyeyGreenstein Method Definitions⟩ +≡
    Float HenyeyGreenstein::Sample_p(const Vector3f &wo, Vector3f *wi,
        const Point2f &u) const {
        ⟨Compute cos θ for Henyey–Greenstein sample 899⟩
        ⟨Compute direction wi for Henyey–Greenstein sample 899⟩
        return PhaseHG(-cosTheta, g);
    }
}

```

For Henyey–Greenstein, the distribution for θ is

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

if $g \neq 0$; otherwise, $\cos \theta = 1 - 2\xi$ gives a uniform sampling over the sphere of directions.

```

⟨Compute cos θ for Henyey–Greenstein sample⟩ ≡
    Float cosTheta;
    if (std::abs(g) < 1e-3)
        cosTheta = 1 - 2 * u[0];
    else {
        Float sqrTerm = (1 - g * g) /
            (1 - g + 2 * g * u[0]);
        cosTheta = (1 + g * g - sqrTerm * sqrTerm) / (2 * g);
    }
}

```

Given the angles $(\cos \theta, \phi)$, what should now be a familiar approach converts them to the direction ω_i .

```

⟨Compute direction wi for Henyey–Greenstein sample⟩ ≡
    Float sinTheta = std::sqrt(std::max((Float)0,
        1 - cosTheta * cosTheta));
    Float phi = 2 * Pi * u[1];
    Vector3f v1, v2;
    CoordinateSystem(wo, &v1, &v2);
    *wi = SphericalDirection(sinTheta, cosTheta, phi, v1, v2, -wo);
}

```

15.3 VOLUMETRIC LIGHT TRANSPORT

CoordinateSystem() 67
 EstimateDirect() 858
 Float 1062
 HenyeyGreenstein::g 682
 PhaseHG() 681
 Pi 1063
 Point2f 68
 SphericalDirection() 346
 Vector3f 60

These sampling building blocks make it possible to implement various light transport algorithms in participating media. We can now implement the fragments in the `EstimateDirect()` function from Section 14.3.1 that handle the cases related to participating media.

First, after a light has been sampled, if the interaction is a scattering event in participating media, it's necessary to compute the value of the phase function for the outgoing direction and the incident illumination direction as well as the value of the PDF for sampling that direction for multiple importance sampling. Because we assume that phase functions are sampled perfectly, these values are the same.

(Evaluate phase function for light sampling strategy) ≡ 859

```
const MediumInteraction &mi = (const MediumInteraction &)it;
Float p = mi.phase->p(mi.wo, wi);
f = Spectrum(p);
scatteringPdf = p;
```

The direct lighting calculation needs to take a sample from the phase function's distribution. `Sample_p()` provides this capability; as described earlier, the value it returns gives both the phase function's value and the PDF's.

(Sample scattered direction for medium interactions) ≡ 860

```
const MediumInteraction &mi = (const MediumInteraction &)it;
Float p = mi.phase->Sample_p(mi.wo, &wi, uScattering);
f = Spectrum(p);
scatteringPdf = p;
```

15.3.1 PATH TRACING

The `VolPathIntegrator` is a `SamplerIntegrator` that accounts for scattering and attenuation from participating media as well as scattering from surfaces. It is defined in the files `integrators/volpath.h` and `integrators/volpath.cpp` and has a general structure that is very similar to the `PathIntegrator`, so here we will only discuss the differences between those two classes. See Figures 15.4 and 15.5 for images rendered with this integrator that show off the importance of accounting for multiple scattering in participating media.

As a `SamplerIntegrator`, the `VolPathIntegrator`'s main responsibility is to implement the `Li()` method. The general structure of its implementation is very similar to that of `PathIntegrator::Li()`, though with a few small changes related to participating media.

(VolPathIntegrator Method Definitions) ≡

```
Spectrum VolPathIntegrator::Li(const RayDifferential &r,
    const Scene &scene, Sampler &sampler, MemoryArena &arena,
    int depth) const {
    Spectrum L(0.f), beta(1.f);
    RayDifferential ray(r);
    bool specularBounce = false;
    for (int bounces = 0; ; ++bounces) {
        <Intersect ray with scene and store intersection in isect 877>
        <Sample the participating medium, if present 901>
        <Handle an interaction with a medium or a surface 902>
        <Possibly terminate the path with Russian roulette 879>
    }
    return L;
}
```

Float 1062
 Interaction::wo 115
 Medium::Sample() 891
 MediumInteraction 688
 MediumInteraction::phase 688
 MemoryArena 1074
 PathIntegrator 875
 PathIntegrator::Li() 876
 PhaseFunction::p() 681
 PhaseFunction::Sample_p() 898
 RayDifferential 75
 Sampler 421
 SamplerIntegrator 25
 Scene 23
 Spectrum 315

At each step in sampling the scattering path, the ray is first intersected with the surfaces in the scene to find the closest surface intersection, if any. Next, participating media are accounted for with a call to the `Medium::Sample()` method, which initializes the provided `MediumInteraction` if a medium interaction should be the next vertex in the path. In



Figure 15.4: Volumetric Path Tracing. (a) Heterogeneous smoke data set rendered with direct lighting only. (b) Rendered with path tracing with a maximum depth of 5. (c) Path tracing with a maximum depth of 25. For this medium, which has an albedo of $\rho = 0.7$, multiple scattering has a significant effect on the final result. For (c), 1024 samples per pixel were required for this noise-free result.

either case, `Sample()` also returns a factor accounting for the beam transmittance and sampling PDF to either the surface or medium interaction.

```

⟨Sample the participating medium, if present⟩ ≡
    MediumInteraction mi;
    if (ray.medium)
        beta *= ray.medium->Sample(ray, sampler, arena, &mi);
        if (beta.IsBlack())
            break;

```

Medium::Sample() 891
MediumInteraction 688
Ray::medium 74
Spectrum::IsBlack() 317

900

In scenes with very dense scattering media, the effort spent on first finding surface intersections will often be wasted, as `Medium::Sample()` will usually generate a medium



Figure 15.5: Homogeneous Volumetric Scattering in Liquid. Scattering in the liquid is modeled with participating media and rendered with the VolPathIntegrator. (Scene courtesy “guismo” from blendswap.com.)

interaction instead. For such scenes, a more efficient implementation would be to first sample a medium interaction, updating the ray’s `tMax` value accordingly before intersecting the ray with primitives in the scene. In turn, surface intersection tests would be much more efficient, as the ray to be tested would often be fairly short. (Further investigating and addressing this issue is left for Exercise 15.5.)

Depending on whether the sampled interaction for this ray is within participating media or at a point on a surface, one of two fragments handles computing the direct illumination at the point and sampling the next direction.

```
(Handle an interaction with a medium or a surface) ≡ 900
    if (mi.IsValid()) {
        ⟨Handle scattering at point in medium for volumetric path tracer 902⟩
    } else {
        ⟨Handle scattering at point on surface for volumetric path tracer⟩
    }
```

Thanks to the fragments defined earlier in this section, the `UniformSampleOneLight()` function already supports estimating direct illumination at points in participating media, so we just need to pass the `MediumInteraction` for the sampled interaction to it. The direction for the ray leaving the medium interaction is then easily found with a call to `Sample_p()`.

```
(Handle scattering at point in medium for volumetric path tracer) ≡ 902
    L += beta * UniformSampleOneLight(mi, scene, arena, sampler, true);
    Vector3f wo = -ray.d, wi;
    mi.phase->Sample_p(wo, &wi, sampler.Get2D());
    ray = mi.SpawnRay(wi);
```

`Interaction::SpawnRay()` 232
`MediumInteraction` 688
`MediumInteraction::IsValid()` 893
`MediumInteraction::phase` 688
`PhaseFunction::Sample_p()` 898
`Sampler::Get2D()` 422
`UniformSampleOneLight()` 856
`Vector3f` 60
`VolPathIntegrator` 900

For scattering from surfaces, the computation performed is almost exactly the same as the regular `PathIntegrator`, except that attenuation of radiance from light sources to surface intersection points is incorporated by calling `VisibilityTester::Tr()` instead of `VisibilityTester::Unoccluded()` when sampling direct illumination. Because these differences are minor, we won't include the corresponding code here.

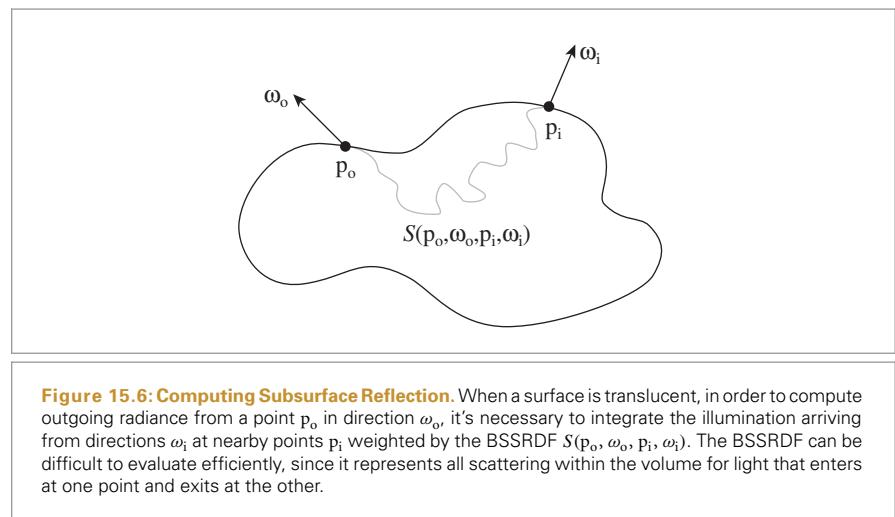
* 15.4 SAMPLING SUBSURFACE REFLECTION FUNCTIONS

We'll now implement techniques to sample the subsurface scattering equation introduced in Section 5.6.2, building on the `BSSRDF` interface introduced in Section 11.4. Our task is to estimate

$$L_o(p_o, \omega_o) = \int_A \int_{\mathcal{H}^2(n)} S(p_o, \omega_o, p_i, \omega_i) L_i(p_i, \omega_i) |\cos \theta_i| d\omega_i dA.$$

Figure 15.6 suggests the complexity of evaluating the integral. To compute the standard Monte Carlo estimate of this equation given a point at which to compute outgoing radiance, we need a technique to sample points p_i on the surface and to compute the incident radiance at these points, as well as an efficient way to compute the specific value of the `BSSRDF` $S(p_o, \omega_o, p_i, \omega_i)$ for each sampled point p_i and incident direction.

The `VolPathIntegrator` could be used to evaluate the `BSSRDF`: given a pair of points on the surface and a pair of directions, the integrator can be used to compute the fraction of incident light from direction ω_i at the point p_i that exits the object at the point p_o in direction ω_o by following light-carrying paths through the multiple scattering events in the medium. Beyond standard path-tracing or bidirectional path-tracing techniques, many other light transport algorithms are applicable to this task.



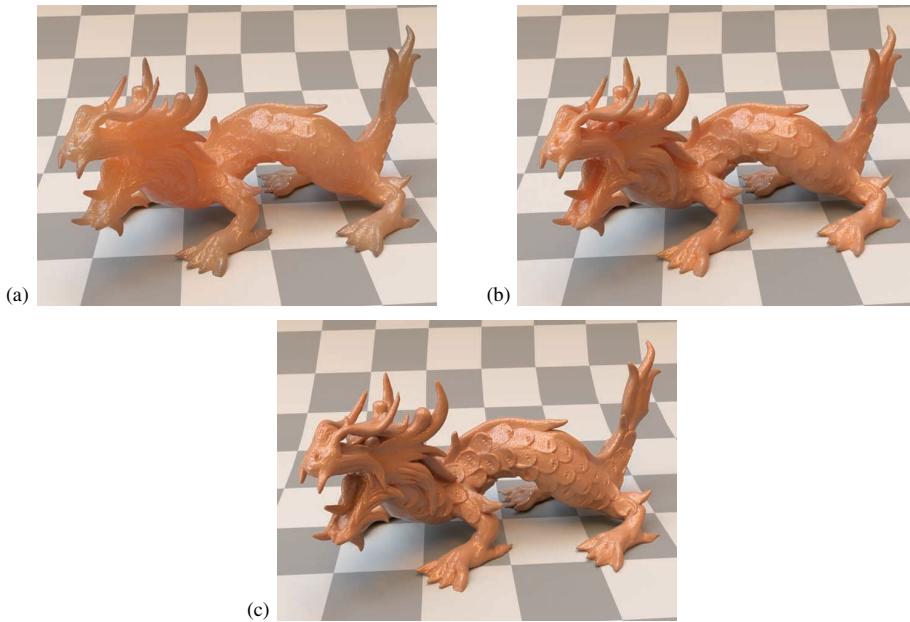


Figure 15.7: Subsurface Scattering from the Dragon Model Rendered Using Different Material Densities.

(a) Although incident illumination is arriving from behind the model, the front of the model has light exiting from it due to subsurface light transport. (b) σ_s and σ_a scaled by a factor of 5. (c) scaled by 25. Note how the dragon becomes increasingly opaque as the scattering coefficients increase.

However, many translucent objects are characterized by having very high albedos, which are not efficiently handled by classic approaches. For example, Jensen et al. (2001b) measured the scattering properties of skim milk and found an albedo of 0.9987. When essentially all of the light is scattered at each interaction in the medium and almost none of it is absorbed, light easily travels far from where it first enters the medium. Hundreds or even thousands of scattering events must be considered to compute an accurate result; given the high albedo of milk, after 100 scattering events, 87.5% of the incident light is still carried by a path, 51% after 500 scattering events, and still 26% after 1000.

BSSRDF class implementations represent the aggregate scattering behavior of these sorts of media, making it possible to render them fairly efficiently. Figure 15.7 shows an example of the dragon model rendered with a BSSRDF. The main sampling operation that must be provided by implementations of the BSSRDF interface, `BSSRDF::Sample_S()`, determines the surface position where a ray re-emerges following internal scattering.

(BSSRDF Interface) +≡ 692
`virtual Spectrum Sample_S(const Scene &scene, Float u1, const Point2f &u2,
 MemoryArena &arena, SurfaceInteraction *si, Float *pdf) const = 0;`

The value of the BSSRDF for the two points and directions is returned directly, and the associated surface intersection record and probability density are returned via the `si` and `pdf` parameters. Two samples must be provided: a 1D sample for discrete sampling decisions (e.g., choosing a specific spectral channel of the profile) and a 2D sample that

`BSSRDF` 692
`BSSRDF::Sample_S()` 904
`Float` 1062
`MemoryArena` 1074
`Point2f` 68
`Scene` 23
`Spectrum` 315
`SurfaceInteraction` 116

is mapped onto si . As we will see shortly, it's useful for BSSRDF implementations to be able to trace rays against the scene geometry to find si , so the scene is also provided as an argument.

15.4.1 SAMPLING THE SeparableBSSRDF

Recall the simplifying assumption introduced in Section 11.4.1, which factored the BSSRDF into spatial and directional components that can be sampled independently from one another. Specifically, Equation (11.6) defined S as a product of a single spatial term and a pair of directional terms related to the incident and outgoing directions.

$$S(\mathbf{p}_o, \omega_o, \mathbf{p}_i, \omega_i) = (1 - F_r(\cos \theta_o)) S_p(\mathbf{p}_o, \mathbf{p}_i) S_\omega(\omega_i). \quad (15.14)$$

The spatial term S_p was further simplified to a radial profile function S_r :

$$S_p(\mathbf{p}_o, \mathbf{p}_i) = S_r(\|\mathbf{p}_o - \mathbf{p}_i\|).$$

We will now explain how each of these factors is handled by the SeparableBSSRDF's sampling routines. This class implements an abstract sampling interface that works for any radial profile function S_r . The TabulatedBSSRDF class, discussed in Section 15.4.2, derives from SeparableBSSRDF and provides a specific tabulated representation of this profile with support for efficient evaluation and exact importance sampling.

Returning to Equation (15.14), if we assume that the BSSRDF is only sampled for rays that are transmitted through the surface boundary, where transmission is selected with probability $(1 - F_r(\cos \theta_o))$, then nothing needs to be done for the $1 - F_r(\cos \theta_o)$ part here. (This is the case for the fragment *<Account for subsurface scattering, if applicable>*.) This is a reasonable expectation to place on calling code, as this approach gives good Monte Carlo efficiency.

```
ARENA_ALLOC() 576
BSDF 572
BxDF 513
Float 1062
Interaction::wo 115
MemoryArena 1074
Point2f 68
Scene 23
SeparableBSSRDF 693
SeparableBSSRDF::Sample_Sp() 908
SeparableBSSRDFAdapter 906
Spectrum 315
Spectrum::IsBlack() 317
SurfaceInteraction 116
SurfaceInteraction::bsdf 250
SurfaceInteraction::shading 118
SurfaceInteraction::shading::n 118
TabulatedBSSRDF 696
Vector3f 60
```

This leaves the S_p and S_ω terms—the former is handled by a call to SeparableBSSRDF::Sample_Sp() (to be discussed shortly), which returns the position si .

(BSSRDF Method Definitions) +≡

```
Spectrum SeparableBSSRDF::Sample_S(const Scene &scene, Float u1,
                                      const Point2f &u2, MemoryArena &arena, SurfaceInteraction *si,
                                      Float *pdf) const {
    Spectrum Sp = Sample_Sp(scene, u1, u2, arena, si, pdf);
    if (!Sp.IsBlack()) {
        (Initialize material model at sampled surface interaction 905)
    }
    return Sp;
}
```

If sampling a position is successful, the method initializes $\text{si} \rightarrow \text{bsdf}$ with an instance of the class SeparableBSSRDFAdapter, which represents the directional term $S_\omega(\omega_i)$ as a BxDF. Although this BxDF does not truly depend on the outgoing direction $\text{si} \rightarrow \text{wo}$, we still need to initialize it with a dummy direction.

(Initialize material model at sampled surface interaction) ≡

```
si->bsdf = ARENA_ALLOC(arena, BSDF)(*si);
si->bsdf->Add(ARENA_ALLOC(arena, SeparableBSSRDFAdapter)(this));
si->wo = Vector3f(si->shading.n);
```

The `SeparableBSSRDFAdapter` class is a thin wrapper around `SeparableBSSRDF::Sw()`. Recall that S_ω from Equation (11.7) was defined as a diffuse-like term scaled by the normalized Fresnel transmission. For this reason, the `SeparableBSSRDFAdapter` classifies itself as `BSDF_DIFFUSE` and just uses the default cosine-weighted sampling routine provided by `BxDF::Sample_f()`.

```
(BSSRDF Declarations) +≡
class SeparableBSSRDFAdapter : public BxDF {
public:
    (SeparableBSSRDFAdapter Public Methods 906)
```

```
private:
    const SeparableBSSRDF *bssrdf;
};
```

(SeparableBSSRDFAdapter Public Methods) ≡

906

```
SeparableBSSRDFAdapter(const SeparableBSSRDF *bssrdf)
    : BxDF(BxDFType(BSDF_REFLECTION | BSDF_DIFFUSE)), bssrdf(bssrdf) {}
```

Similar to refractive BSDFs, a scaling factor related to the light transport mode must be applied to the value the `f()` method returns for the S_ω term. This issue is discussed in more detail in Section 16.1, and the fragment that applies this scaling, *(Update BSSRDF transmission term to account for adjoint light transport)*, is defined there.

(SeparableBSSRDFAdapter Public Methods) +≡

906

```
Spectrum f(const Vector3f &wo, const Vector3f &wi) const {
    Spectrum f = bssrdf->Sw(wi);
    (Update BSSRDF transmission term to account for adjoint light transport 961)
    return f;
}
```

To sample the spatial component S_p , we need a way of mapping a 2D distribution function onto an arbitrary surface using a parameterization of the surface in the neighborhood of the outgoing position. A conceptually straightforward way to obtain such a parameterization is by means of geodesics, but finding and evaluating them is non-trivial and requires significant implementation effort for each shape that is supported. We use much a simpler approach that uses ray tracing to map the radial profile S_r onto the scene geometry.

Figure 15.8 illustrates the basic idea: the position p_o and associated normal n_o define a planar approximation to the surface. Using 2D polar coordinates, we first sample an azimuth ϕ and a radius value r centered around p_o and then map this position onto the actual surface by intersecting an offset perpendicular ray with the primitive, producing the position p_i . The `SeparableBSSRDF` class only supports radially symmetric profile functions; hence ϕ is drawn from a uniform distribution on $[0, 2\pi]$, and r is distributed according to the radial profile function S_r .

There are still several difficulties with this basic approach:

- The radial profile S_r is not necessarily uniform across wavelengths—in practice, the mean free path can differ by orders of magnitude between different spectral channels.

BSDF_DIFFUSE 513
 BSDF_REFLECTION 513
 BxDF 513
 BxDF::Sample_f() 806
 BxDFType 513
 SeparableBSSRDF 693
 SeparableBSSRDF::Sw() 695
 SeparableBSSRDFAdapter 906
 SeparableBSSRDFAdapter::
 bssrdf
 906
 Spectrum 315
 Vector3f 60

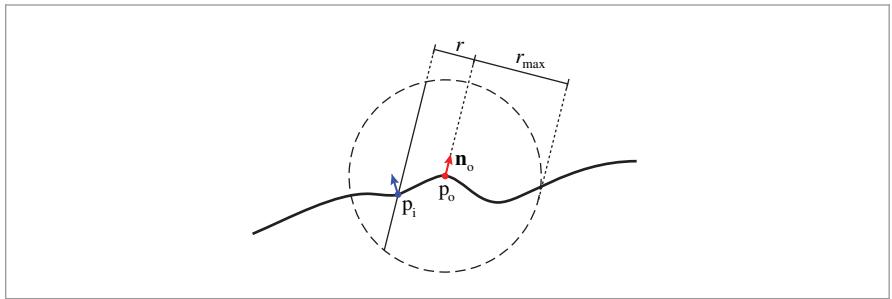


Figure 15.8: Sampling the Spatial Component of the Separable BSSRDF. To compute the outgoing radiance at a point p_o on a translucent surface, we sample a radius r from a radial scattering profile and map it onto the surface by tracing a probe ray back toward the surface, in the opposite direction of the surface normal n_o . To improve efficiency, the probe ray is clamped to a sphere of radius r_{\max} , after which the value of the BSSRDF becomes negligible.

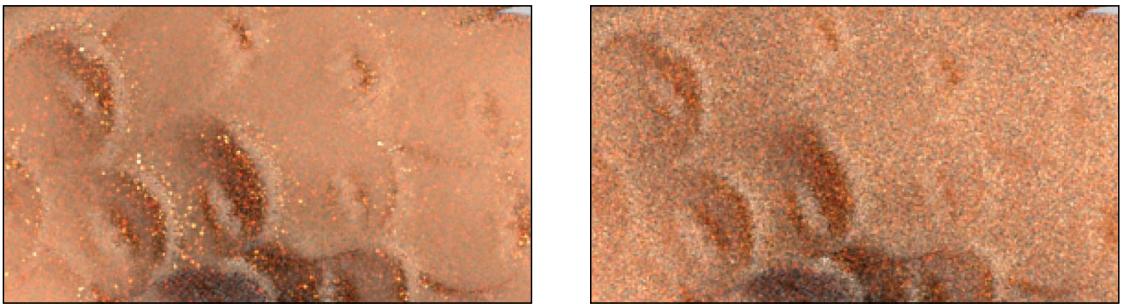


Figure 15.9: Comparison of Scattering Profile Projection. (a) Projecting the scattering profile perpendicularly along the current normal direction usually works well, but occasionally there are surface regions that are sampled with a very small probability despite a large corresponding BSSRDF value S , producing high variance in renderings. (b) Projecting along multiple axes and combining the resulting sampling techniques using multiple importance sampling greatly reduces the maximum variance at the cost of an increase in the overall amount of variance in well-converged regions. As more samples are taken, the latter approach shows better overall convergence.

- If the surface geometry is poorly approximated by a plane and $\mathbf{n}_o \cdot \mathbf{n}_i \approx 0$, where \mathbf{n}_i is the surface normal at p_i , the probe rays will hit the surface at a grazing angle so that positions p_i with comparatively high values of $S(p_o, \omega_o, p_i, \cdot)$ may be sampled with too low a probability. The result is high variance in renderings (Figure 15.9).
- Finally, the probe ray may intersect multiple surface locations along its length, all of which may contribute to reflected radiance.

The first two problems can be addressed with a familiar approach; namely, by introducing additional tailored sampling distributions and combining them using multiple importance sampling. The third will be addressed shortly.

We use a different sampling technique per wavelength to deal with spectral variation, and each technique is additionally replicated three times with different projection axes given by the basis vectors of a local frame, resulting in a total of $3 * \text{Spectrum}::nSamples$

sampling techniques. This ensures that every point where S takes on non-negligible values is intersected with a reasonable probability. This combination of techniques is implemented in `SeparableBSSRDF::Sample_Sp()`.

```
(BSSRDF Method Definitions) +≡
Spectrum SeparableBSSRDF::Sample_Sp(const Scene &scene, Float u1,
    const Point2f &u2, MemoryArena &arena, SurfaceInteraction *pi,
    Float *pdf) const {
    (Choose projection axis for BSSRDF sampling 908)
    (Choose spectral channel for BSSRDF sampling 909)
    (Sample BSSRDF profile in polar coordinates 909)
    (Compute BSSRDF profile bounds and intersection height 909)
    (Compute BSSRDF sampling ray segment 910)
    (Intersect BSSRDF sampling ray against the scene geometry 910)
    (Randomly choose one of several intersections during BSSRDF sampling 911)
    (Compute sample PDF and return the spatial BSSRDF term  $S_p$  912)
}
```

We begin by choosing a projection axis. Note that when the surface is close to planar, projecting along the normal `SeparableBSSRDF::ns` is clearly the best sampling strategy, as probe rays along the other two axes are likely to miss the surface. We therefore allocate a fairly large portion (50%) of the sample budget to perpendicular rays. The other half is equally shared between tangential projections along `SeparableBSSRDF::ss` and `SeparableBSSRDF::ts`. The three axes of the chosen coordinate system are stored in `vx`, `vy`, and `vz`, and follow our usual convention of measuring angles θ in spherical coordinates with respect to the z axis.

After this discrete sampling operation, we scale and offset `u1` so that additional sampling operations can reuse it as a uniform variate.

```
(Choose projection axis for BSSRDF sampling) ≡ 908
Vector3f vx, vy, vz;
if (u1 < .5f) {
    vx = ss;
    vy = ts;
    vz = Vector3f(ns);
    u1 *= 2;
} else if (u1 < .75f) {
    (Prepare for sampling rays with respect to ss)
} else {
    (Prepare for sampling rays with respect to ts)
}
```

Float 1062
MemoryArena 1074
Point2f 68
Scene 23
`SeparableBSSRDF::ns` 693
`SeparableBSSRDF::Sample_Sp()` 908
`SeparableBSSRDF::ss` 693
`SeparableBSSRDF::ts` 693
Spectrum 315
SurfaceInteraction 116
Vector3f 60

The fragments for the other two axes are similar and therefore not included here.

Next, we uniformly choose a spectral channel and re-scale `u1` once more.

```
(Choose spectral channel for BSSRDF sampling) ≡ 908
    int ch = Clamp((int)(u1 * Spectrum::nSamples),
                    0, Spectrum::nSamples - 1);
    u1 = u1 * Spectrum::nSamples - ch;
```

The 2D profile sampling operation is then carried out in polar coordinates using the SeparableBSSRDF::Sample_Sr() method. This method returns a negative radius to indicate a failure (e.g., when there is no scattering from channel ch); the implementation here returns a BSSRDF value of 0 in this case.

```
(Sample BSSRDF profile in polar coordinates) ≡ 908
    Float r = Sample_Sr(ch, u2[0]);
    if (r < 0)
        return Spectrum(0.f);
    Float phi = 2 * Pi * u2[1];
```

Both the radius sampling method SeparableBSSRDF::Sample_Sr() and its associated density function SeparableBSSRDF::Pdf_Sr() are declared as pure virtual functions; an implementation for TabulatedBSSRDF is presented in the next section.

```
(SeparableBSSRDF Interface) +≡ 693
    virtual Float Sample_Sr(int ch, Float u) const = 0;
    virtual Float Pdf_Sr(int ch, Float r) const = 0;
```

Because the profile falls off fairly quickly, we are not interested in positions p_i that are too far² away from p_o . In order to reduce the computational expense of the ray-tracing step, the probe ray is clamped to a sphere of radius r_{\max} around p_o . Another call to SeparableBSSRDF::Sample_Sr() is used to determine r_{\max} . Assuming that this function implements a perfect importance sampling scheme based on the inversion method (Section 13.3.1), Sample_Sr() maps a sample value x to the radius of a sphere containing a fraction x of the scattered energy.

Here, we set r_{\max} so that the sphere from Figure 15.8 contains 99.9% of the scattered energy. When r lies outside of r_{\max} , sampling fails—this helps to keep the probe rays short, which significantly improves the run-time performance. Given r and r_{\max} , the length of the intersection of the probe ray with the sphere of radius r_{\max} is

$$l = 2\sqrt{r_{\max}^2 - r^2}.$$

Clamp() 1062

CoefficientSpectrum::nSamples
318

Float 1062

Pi 1063

SeparableBSSRDF::Pdf_Sr()
909

SeparableBSSRDF::Sample_Sr()
909

Spectrum 315

TabulatedBSSRDF 696

(See Figure 15.10.)

```
(Compute BSSRDF profile bounds and intersection height) ≡ 908
    Float rMax = Sample_Sr(ch, 0.999f);
    if (r > rMax)
        return Spectrum(0.f);
    Float l = 2 * std::sqrt(rMax * rMax - r * r);
```

2 This assumption can be problematic when a material is illuminated by a very bright light source (e.g., a hand held in front of a flashlight), in which case long-range light transport remains important.

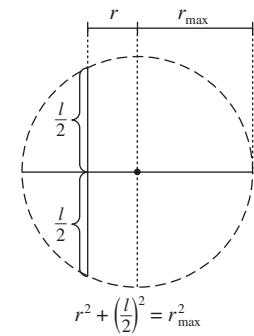


Figure 15.10: Given a sampled radius r that is less than the maximum radius r_{\max} , the length of the segment l , which is the total length of the ray in the sphere, can be found using the Pythagorean theorem.

Given the sampled polar coordinate value, we can compute the world space origin of a ray that lies on the boundary of the sphere and a target point, $pTarget$, where it exits the sphere.

(Compute BSSRDF sampling ray segment) ≡

908

```
Interaction base;
base.p = po.p + r * (vx * std::cos(phi) + vy * std::sin(phi)) -
    1 * vz * 0.5f;
base.time = po.time;
Point3f pTarget = base.p + 1 * vz;
```

In practice, there could be more than just one intersection along the probe ray, and we want to collect all of them here. We'll create a linked list of all of the found interactions.

(Intersect BSSRDF sampling ray against the scene geometry) ≡

908

```
(Declare IntersectionChain and linked list 910)
(Accumulate chain of intersections along ray 911)
```

`IntersectionChain` lets us maintain this list. Once again, the `MemoryArena` makes it possible to efficiently perform allocations, here for the list nodes.

(Declare IntersectionChain and linked list) ≡

910

```
struct IntersectionChain {
    SurfaceInteraction si;
    IntersectionChain *next = nullptr;
};
IntersectionChain *chain = ARENA_ALLOC(arena, IntersectionChain)();
```

ARENA_ALLOC() 576
Interaction 115
Interaction::p 115
Interaction::time 115
IntersectionChain 910
MemoryArena 1074
Point3f 68
SurfaceInteraction 116

We now start by finding intersections along the segment within the sphere. The list's tail node's `SurfaceInteraction` is initialized with each intersection's information, and `base` `Interaction` is updated so that the next ray can be spawned on the other side of the intersected surface. (See Figure 15.11.)

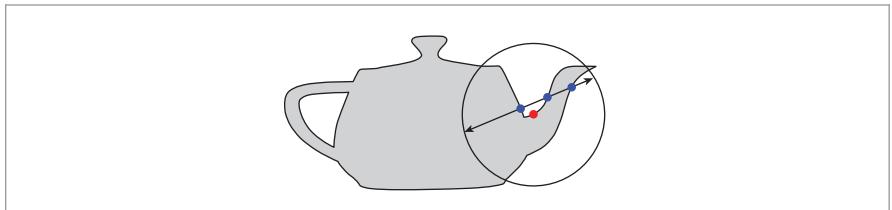


Figure 15.11: Accumulating Surface Intersections along a Sample Ray. The SeparableBSSRDF::Sample_Sp() method finds all of the intersections of a ray with the surface of the primitive, where ray extents are limited to a sphere around the intersection point (red dot). At each intersection (blue dots), the corresponding SurfaceInteraction is stored in a linked list before a new ray leaving the other side of the intersected surface is generated.

(Accumulate chain of intersections along ray) ≡

910

```
IntersectionChain *ptr = chain;
int nFound = 0;
while (scene.Intersect(base.SpawnRayTo(pTarget), &ptr->si)) {
    base = ptr->si;
    (Append admissible intersection to IntersectionChain 911)
}
```

When tracing rays to sample nearby points on the surface of the primitive, it's important to ignore any intersections on other primitives in the scene. (There is an implicit assumption that scattering between primitives will be handled by the integrator, and the BSSRDF should be limited to account for single primitives' scattering.) The implementation here uses equality of Material pointers as a proxy to determine if an intersection is on the same primitive. Valid intersections are appended to the chain, and the variable nFound records their total count when the loop terminates.

(Append admissible intersection to IntersectionChain) ≡

911

```
ARENA_ALLOC() 576
Interaction::SpawnRayTo() 232
IntersectionChain 910
IntersectionChain::next 910
IntersectionChain::si 910
Material 577
Primitive::GetMaterial() 249
Scene::Intersect() 24
SeparableBSSRDF::material 693
SeparableBSSRDF::Sample_Sp() 908
Spectrum 315
SurfaceInteraction 116
SurfaceInteraction::primitive 249
```

With the set of intersections at hand, we must now choose one of them, as Sample_Sp() can only return a single position p_i . The following fragment uses the variable u_1 one last time to pick one of the list entries with uniform probability.

(Randomly choose one of several intersections during BSSRDF sampling) ≡

908

```
if (nFound == 0)
    return Spectrum(0.0f);
int selected = Clamp((int)(u1 * nFound), 0, nFound - 1);
while (selected-- > 0)
    chain = chain->next;
*pi = chain->si;
```

Finally, we can call `SeparableBSSRDF::Pdf_Sp()` (to be defined shortly) to evaluate the combined PDF that takes all of the sampling strategies into account. The probability it returns is divided by `nFound` to account for the discrete probability of selecting p_i from the `IntersectionChain`. Finally, the value of $S_p(p_i)$ is returned.

(Compute sample PDF and return the spatial BSSRDF term S_p) ≡ 908

```
*pdf = Pdf_Sp(*pi) / nFound;
return Sp(*pi);
```

`SeparableBSSRDF::Pdf_Sp()` returns the probability per unit area of sampling the position p_i with the total of $3 * \text{Spectrum}::nSamples$ sampling techniques available to `SeparableBSSRDF::Sample_Sp()`.

(BSSRDF Method Definitions) +≡

```
Float SeparableBSSRDF::Pdf_Sp(const SurfaceInteraction &pi) const {
    <Express  $p_i - p_o$  and  $n_i$  with respect to local coordinates at  $p_o$  912>
    <Compute BSSRDF profile radius under projection along each axis 913>
    <Return combined probability from all BSSRDF sampling strategies 913>
}
```

First, `nLocal` is initialized with the surface normal at p_i and `dLocal` with the difference vector $p_o - p_i$, both expressed using local coordinates at p_o .

(Express $p_i - p_o$ and n_i with respect to local coordinates at p_o) ≡ 912

```
Vector3f d = po.p - pi.p;
Vector3f dLocal(Dot(ss, d), Dot(ts, d), Dot(ns, d));
Normal3f nLocal(Dot(ss, pi.n), Dot(ts, pi.n), Dot(ns, pi.n));
```

To determine the combined PDF, we must query the probability of sampling a radial profile radius matching the pair (p_o, p_i) for each technique. This radius is measured in 2D and thus depends on the chosen projection axis (Figure 15.12). The `rProj` variable records radii for projections perpendicular to `ss`, `ts`, and `ns`.

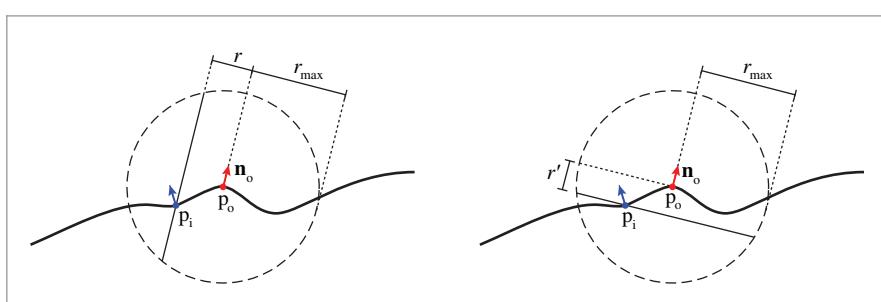


Figure 15.12: Probability of Alternative Sampling Strategies. To determine the combined probability of a BSSRDF position sample p_i (left), we must evaluate the radial PDF for each of the radii r' corresponding to alternative projection axes (right).

```
Dot() 63
Float 1062
Interaction::n 116
Interaction::p 115
Normal3f 71
SeparableBSSRDF::ns 693
SeparableBSSRDF::Pdf_Sp() 912
SeparableBSSRDF::Sp() 695
SeparableBSSRDF::ss 693
SeparableBSSRDF::ts 693
SurfaceInteraction 116
Vector3f 60
```

```
(Compute BSSRDF profile radius under projection along each axis) ≡ 912
    Float rProj[3] = { std::sqrt(dLocal.y * dLocal.y + dLocal.z * dLocal.z),
                      std::sqrt(dLocal.z * dLocal.z + dLocal.x * dLocal.x),
                      std::sqrt(dLocal.x * dLocal.x + dLocal.y * dLocal.y) };
```

The remainder of the implementation simply loops over all combinations of spectral channels and projection axes and sums up the product of the probability of selecting each technique and its area density under projection onto the surface at p_0 .

```
(Return combined probability from all BSSRDF sampling strategies) ≡ 912
    Float pdf = 0, axisProb[3] = { .25f, .25f, .5f };
    Float chProb = 1 / (Float)Spectrum::nSamples;
    for (int axis = 0; axis < 3; ++axis)
        for (int ch = 0; ch < Spectrum::nSamples; ++ch)
            pdf += Pdf_Sr(ch, rProj[axis]) * std::abs(nLocal[axis]) *
                chProb * axisProb[axis];
    return pdf;
```

The alert reader may have noticed a slight inconsistency in the above definitions: the probability of choosing one of several (`nFound`) intersections in `SeparableBSSRDF::Sample_Sp()` should really have been part of the density function computed in the `SeparableBSSRDF::Pdf_Sp()` method rather than the ad hoc division that occurs in the fragment *(Compute sample PDF and return the spatial BSSRDF term S_p)*. In practice, the number of detected intersections varies with respect of the projection axis and spectral channel; correctly accounting for this in the PDF computation requires counting the number of intersections along each of a total of $3 * \text{Spectrum}::\text{nSamples}$ probe rays for every sample! We neglect this issue, trading a more efficient implementation for a small amount of bias.

15.4.2 SAMPLING THE TabulatedBSSRDF

```
CoefficientSpectrum::nSamples 318
Float 1062
FourierBSDF 555
FourierBSDF::Sample_f() 819
SampleCatmullRom2D() 824
SeparableBSSRDF 693
SeparableBSSRDF::Pdf_Sp() 912
SeparableBSSRDF::Sample_Sp() 908
Spectrum 315
TabulatedBSSRDF 696
TabulatedBSSRDF::Pdf_Sr() 914
TabulatedBSSRDF::Sample_Sr() 914
```

The previous section completed the discussion of BSSRDF sampling with the exception of the `Pdf_Sr()` and `Sample_Sr()` methods that were declared as pure virtual functions in the `SeparableBSSRDF` interface. The `TabulatedBSSRDF` subclass implements this missing functionality.

The `TabulatedBSSRDF::Sample_Sr()` method samples radius values proportional to the radial profile function S_r . Recall from Section 11.4.2 that the profile has an implicit dependence on the albedo ρ at the current surface position and that the `TabulatedBSSRDF` provides interpolated evaluations of $S_r(\rho, r)$ using 2D tensor product spline basis functions. `TabulatedBSSRDF::Sample_Sr()` then determines the albedo ρ for the given spectral channel ch and draws samples proportional to the remaining 1D function $S_r(\rho, \cdot)$. Sample generation fails if there is neither scattering nor absorption on channel ch (this case is indicated by returning a negative radius).

As in Section 11.4.2, there is a considerable amount of overlap with the `FourierBSDF` implementation. The sampling operation here actually reduces to a single call to `SampleCatmullRom2D()`, which was previously used in `FourierBSDF::Sample_f()`.

```
(BSSRDF Method Definitions) +≡
Float TabulatedBSSRDF::Sample_Sr(int ch, Float u) const {
    if (sigma_t[ch] == 0)
        return -1;
    return SampleCatmullRom2D(table.nRhoSamples, table.nRadiusSamples,
        table.rhoSamples.get(), table.radiusSamples.get(),
        table.profile.get(), table.profileCDF.get(),
        rho[ch], u) / sigma_t[ch];
}
```

Recall that this function depends on a precomputed CDF array, which is initialized when the BSSRDFTable is created.

(BSSRDFTable Public Data) +≡
`std::unique_ptr<Float[]> profileCDF;`

697

The Pdf_Sr() method returns the PDF of samples obtained via Sample_Sr(). It evaluates the profile function divided by the normalizing constant ρ_{eff} defined in Equation (11.11).

The beginning is analogous to the spline evaluation code in TabulatedBSSRDF::Sr(). The fragment *(Compute spline weights to interpolate BSSRDF density on channel ch)* matches *(Compute spline weights to interpolate BSSRDF on channel ch)* in that method except that this method immediately returns zero if the optical radius is outside the range represented by the spline.

```
(BSSRDF Method Definitions) +≡
Float TabulatedBSSRDF::Pdf_Sr(int ch, Float r) const {
    (Convert r into unitless optical radius roptical 699)
    (Compute spline weights to interpolate BSSRDF density on channel ch)
    (Return BSSRDF profile density for channel ch 914)
}
```

The remainder of the implementation is very similar to fragment *(Set BSSRDF value Sr[ch] using tensor spline interpolation)* except that here, we also interpolate ρ_{eff} from the tabulation and include it in the division at the end.

```
(Return BSSRDF profile density for channel ch) ≡
914
Float sr = 0, rhoEff = 0;
for (int i = 0; i < 4; ++i) {
    if (rhoWeights[i] == 0) continue;
    rhoEff += table.rhoEff[rhoOffset + i] * rhoWeights[i];
    for (int j = 0; j < 4; ++j) {
        if (radiusWeights[j] == 0) continue;
        sr += table.EvalProfile(rhoOffset + i, radiusOffset + j) *
            rhoWeights[i] * radiusWeights[j];
    }
}
(Cancel marginal PDF factor from tabulated BSSRDF profile 700)
return std::max((Float)0, sr * sigma_t[ch] * sigma_t[ch] / rhoEff);
```

BSSRDFTable 697
BSSRDFTable::EvalProfile() 700
BSSRDFTable::nRadiusSamples 698
BSSRDFTable::nRhoSamples 698
BSSRDFTable::profile 698
BSSRDFTable::profileCDF 914
BSSRDFTable::radiusSamples 698
BSSRDFTable::rhoEff 698
BSSRDFTable::rhoSamples 698
Float 1062
SampleCatmullRom2D() 824
TabulatedBSSRDF 696
TabulatedBSSRDF::rho 697
TabulatedBSSRDF::sigma_t 697
TabulatedBSSRDF::Sr() 699

15.4.3 SUBSURFACE SCATTERING IN THE PATH TRACER

We now have the capability to apply Monte Carlo integration to the generalized scattering equation, (5.11), from Section 5.6.2. We will compute estimates of the form

$$L_o(p_o, \omega_o) \approx \frac{S(p_o, \omega_o, p_i, \omega_i) (L_d(p_i, \omega_i) + L_i(p_i, \omega_i)) |\cos \theta_i|}{p(p_i) p(\omega_i)},$$

where L_d represents incident direct radiance and L_i is incident indirect radiance. The sample (p_i, ω_i) is generated in two steps. First, given p_o and ω_o , a call to `BSSRDF::Sample_S()` returns a position p_i whose distribution is similar to the marginal distribution of S with respect to p_i .

Next, we sample the incident direction ω_i . Recall that the `BSSRDF::Sample_S()` interface intentionally keeps these two steps apart: instead of generating both p_i and ω_i at the same time, it returns a special BSDF instance via the `bsdf` field of `si` that is used for the direction sampling step. No generality is lost with such an approach: the returned BSDF can be completely arbitrary and is explicitly allowed to depend on information computed within `BSSRDF::Sample_S()`. The benefit is that we can re-use a considerable amount of existing infrastructure for computing product integrals of a BSDF and L_i .

The PathIntegrator's *(Find next path vertex and accumulate contribution)* fragment invokes the following code to compute this estimate.

```
<Account for subsurface scattering, if applicable> ≡ 877
    if (isect.bssrdf && (flags & BSDF_TRANSMISSION)) {
        <Importance sample the BSSRDF 915>
        <Account for the direct subsurface scattering component 915>
        <Account for the indirect subsurface scattering component 916>
    }
```

When the BSSRDF sampling case is triggered, the path tracer begins by calling `BSSRDF::Sample_S()` to generate p_i and incorporates the resulting sampling weight into its throughput weight variable `beta` upon success.

```
<Importance sample the BSSRDF> ≡ 915
    SurfaceInteraction pi;
    Spectrum S = isect.bssrdf->Sample_S(scene, sampler.Get1D(),
                                              sampler.Get2D(), arena, &pi, &pdf);
    if (S.IsBlack() || pdf == 0)
        break;
    beta *= S / pdf;

BSDF 572
BSDF_TRANSMISSION 513
BSSRDF::Sample_S() 904
PathIntegrator 875
Sampler::Get1D() 422
Sampler::Get2D() 422
Spectrum 315
Spectrum::IsBlack() 317
SurfaceInteraction 116
SurfaceInteraction::bssrdf 250
UniformSampleOneLight() 856
```

Because `BSSRDF::Sample_S()` also initializes `pi`'s `bsdf` with a BSDF that characterizes the dependence of S on ω_i , we can reuse the existing infrastructure for direct illumination computations. Only a single line of code is necessary to compute the contribution of direct lighting at p_i to reflected radiance at p_o .

```
<Account for the direct subsurface scattering component> ≡ 915
    L += beta * UniformSampleOneLight(pi, scene, arena, sampler);
```



Figure 15.13: Subsurface Scattering with the PathIntegrator. These dragons both have BSSRDFs that describe subsurface scattering in their interiors. (Model courtesy of Christian Schüller.)

Similarly, accounting for indirect illumination at the newly sampled incident point is almost the same as the BSDF indirect illumination computation in the PathIntegrator except that pi is used for the next path vertex instead of isect .

(Account for the indirect subsurface scattering component) \equiv 915

```
Spectrum f = pi.bsdf->Sample_f(pi.wo, &wi, sampler.Get2D(), &pdf,
                                BSDF_ALL, &flags);
if (f.IsBlack() || pdf == 0)
    break;
beta *= f * AbsDot(wi, pi.shading.n) / pdf;
specularBounce = (flags & BSDF_SPECULAR) != 0;
ray = pi.SpawnRay(wi);
```

With this, the path tracer (and the volumetric path tracer) support subsurface scattering. See Figure 15.13 for an example.

AbsDot() 64
BSDF::Sample_f() 832
BSDF_ALL 513
BSDF_SPECULAR 513
Interaction::SpawnRay() 232
Interaction::wo 115
PathIntegrator 875
Sampler::Get2D() 422
Spectrum 315
Spectrum::IsBlack() 317
SurfaceInteraction::bsdf 250
SurfaceInteraction::shading
118
SurfaceInteraction::
shading::n
118

* 15.5 SUBSURFACE SCATTERING USING THE DIFFUSION EQUATION

Our last task to complete the subsurface scattering implementation is to be able to initialize the TabulatedBSSRDF with a radial profile function S_r that accurately describes subsurface scattering for given properties of the scattering medium (σ_a , σ_s , the phase function asymmetry parameter g , and the relative index of refraction η). The technique we'll discuss in this section is based on the *photon beam diffusion* (PBD) technique by

Habel et al. (2013). The resulting profile takes all orders of scattering into account, effectively accounting for all of the light transport that occurs within the surface.

Beam diffusion makes several significant assumptions and approximations: first, the distribution of light in the translucent medium is modeled with the *diffusion approximation*, which describes the equilibrium distribution of illumination in highly scattering optically thick participating media. Second, it assumes homogeneous scattering properties throughout the medium, and it implicitly assumes that the medium is *semi-infinite* (it continues infinitely beneath a planar surface of infinite lateral extent). Finally, PBD builds upon the separable BSSRDF approximation of Equation (11.6), which imposes a simple multiplicative relationship between the spatial and directional scattering distribution. When these approximations are satisfied, the solutions computed by PBD are in close agreement with ground truth simulations performed using the equation of transfer, Equation (15.2).

Of course, many of these assumptions won't be valid when the profile is applied to an arbitrary shape, potentially also with spatially varying material properties. The appeal of diffusion-type methods in the context of computer graphics is that they degrade in a graceful manner, producing visually reasonable results even in cases where some or all of their fundamental assumptions are violated. See the “Further Reading” section and exercises at the end of this chapter for references to improvements to this approach that generalize it to handle a wider range of settings more accurately.

While PBD can compute the profile S_r for any radius and material parameters, profile evaluations tend to be fairly expensive, as they involve a numerical integration step. Furthermore, we need to be able to invert the CDF of S_r in polar coordinates to importance sample the model, but this inverse is not available in closed form. The TabulatedBSSRDF from Section 11.4.2 nicely addresses these issues, providing efficient evaluation and sampling operations. Our approach will thus be to precompute S_r for a range of radii and albedo values and use the results to populate the BSSRDFTable of a TabulatedBSSRDF. This precomputation is performed during scene construction.

In the following, we discuss the key ingredients of the PBD method, starting with the principle of similarity and diffusion theory.

15.5.1 PRINCIPLE OF SIMILARITY

A number of important ideas are used in the process of transforming the fully general equation of transfer to the diffusion equation, which can be approximately solved for subsurface scattering. The first is the *principle of similarity*, which says that for an anisotropically scattering medium with a high albedo, the medium can instead be modeled as having an isotropic phase function with appropriately modified scattering and attenuation coefficients. Light transport solutions computed based on the modified coefficients correspond well to those with the original coefficients and phase function, while allowing simplifications due to the assumption of isotropic scattering.

BSSRDFTable 697
TabulatedBSSRDF 696

The principle of similarity is based on the observation that after many scattering events the distribution of light in media with high albedos becomes more and more uniformly directionally distributed regardless of the original illumination distribution or the anisotropy of the phase function. One way to see how this happens is to consider

an expression derived by Yanovitskij (1997); it describes isotropization due to multiple scattering events from the Henyey–Greenstein phase function. He showed that the distribution of light that has been scattered n times is given by

$$p_n(\omega \rightarrow \omega') = \frac{1 - g^{2n}}{4\pi(1 + g^{2n} - 2g|g^{n-1}|(-\omega \cdot \omega')^{3/2})}.$$

As n grows large, this converges to the isotropic phase function, $1/(4\pi)$. Figure 15.14 plots this function for a few values of n . When coupled with the observation from Section 15.4.1 about how much of the light energy remains after tens or even hundreds of scattering events in high-albedo materials, one can see intuitively why it is reasonable to work with an isotropic phase function approximation for high-albedo media.

If the principle of similarity is applied to treat the phase function as isotropic, modified versions of various scattering properties should be used. The *reduced scattering coefficient* is defined as $\sigma'_s = (1 - g)\sigma_s$, and the *reduced attenuation coefficient* is $\sigma'_t = \sigma_a + \sigma'_s$. The albedo also changes to a *reduced albedo* defined as $\rho' = \sigma'_s/\sigma'_t$, which is generally different from ρ . These new coefficients account for the effect of using the isotropic phase function approximation.

To understand the idea these coefficients embody, consider a strongly forward-scattering phase function, where $g \rightarrow 1$. With the original phase function, light will mostly continue in the same direction once it scatters. In this example, the value of the reduced scattering coefficient $\sigma'_s = (1 - g)\sigma_s$ is much smaller than σ_s , which means that light travels a larger distance in the medium before scattering; the medium is approximated as being thinner, allowing light to travel farther. The change thus has the same effect as a highly forward-scattering phase function.

Conversely, consider the case of $g \rightarrow -1$. In this case, at a scattering event the light will tend to scatter back in the direction it came from. But then the next time it scatters after that, it will generally reverse course again; it bounces back and forth without making very much forward progress. In this case, the reduced scattering coefficient is larger than the original scattering coefficient, indicating greater probability of a scattering interaction. In other words, the medium is treated as being thicker than it actually is, which approximates the effect of light having relatively more trouble making forward progress. Figure 15.15 illustrates these ideas, showing representative paths of scattering interactions in highly forward-scattering and highly backward-scattering media.

15.5.2 DIFFUSION THEORY

Diffusion theory provides a way of transitioning from the equation of transfer to a simpler *diffusion equation*, which provides a solution to the equation of transfer for the case of homogeneous, optically thick, highly scattering materials (i.e., those with relatively large albedos). For the application to subsurface scattering in pbrt, it can be derived by writing the equation of transfer using the reduced scattering and attenuation coefficients and an isotropic phase function.

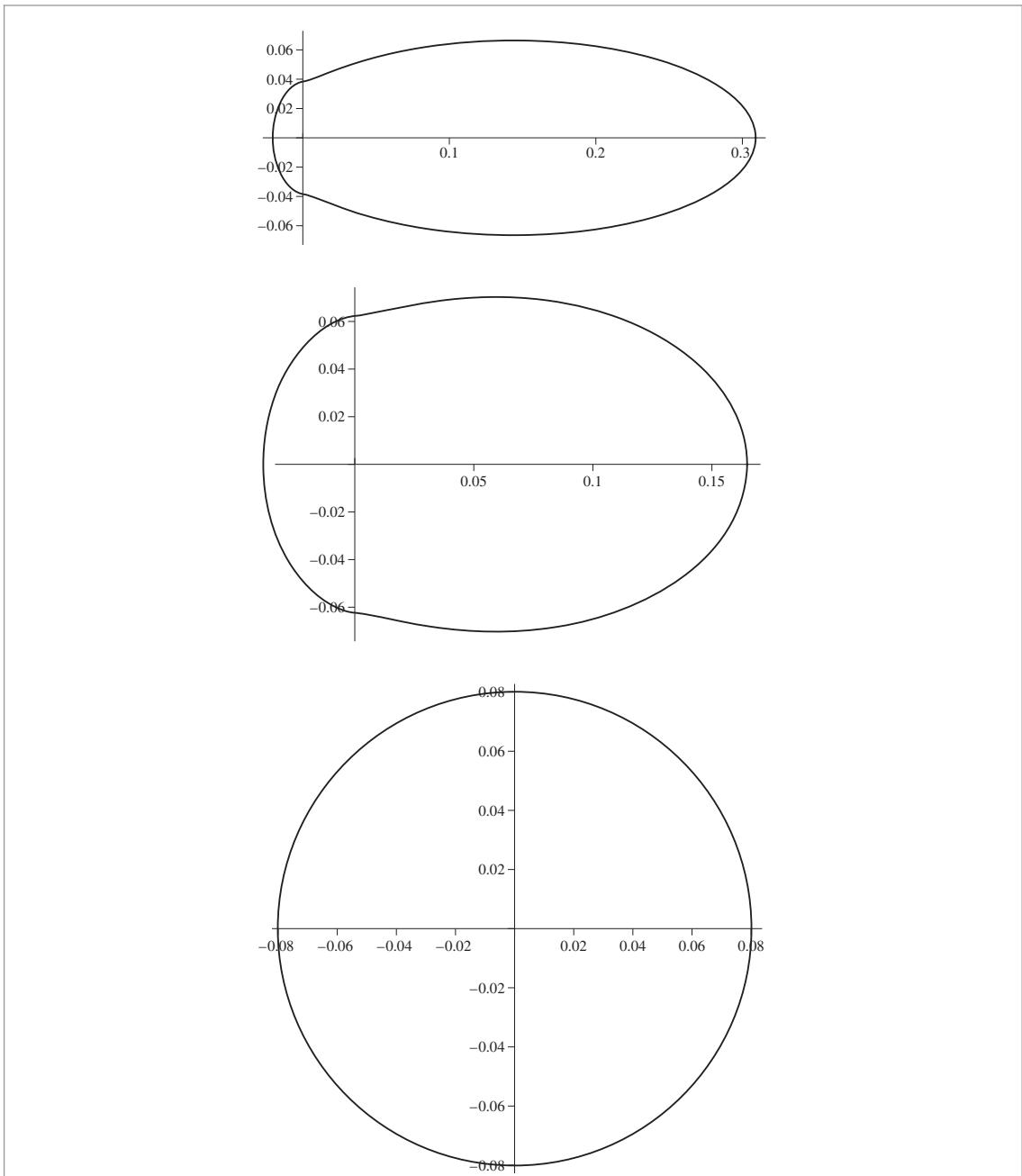


Figure 15.14: Light Distribution after Many Scattering Events. (a) Directional distribution of a single incident ray of light after 10 scattering events in a highly anisotropic medium with $g = 0.9$, (b) 100 scattering events, (c) 1000 scattering events. The distribution becomes increasingly isotropic, even though it was initially very anisotropic.

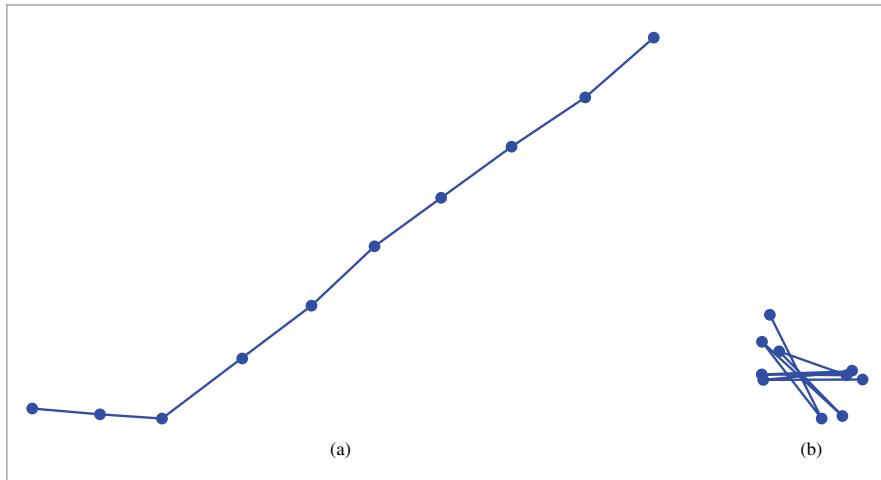


Figure 15.15: Representative Light Paths for Highly Anisotropic Scattering Media. (a) Forward-scattering medium, with $g = 0.9$. Light generally scatters in the same direction it was originally traveling. (b) Backward-scattering medium, with $g = -0.9$. Light frequently bounces back and forth, making relatively little forward progress with respect to its original direction.

Starting with the integro-differential form of the equation of transfer from Equation (15.1),

$$\begin{aligned} \frac{\partial}{\partial t} L_o(p + t\omega, \omega) &= -\sigma_t(p, \omega)L_i(p, -\omega) \\ &\quad + \sigma_s(p, \omega) \int_{S^2} p(p, -\omega', \omega)L_i(p, \omega') d\omega' + L_e(p, \omega), \end{aligned}$$

we assume spatially uniform material parameters and switch to an isotropic phase function $p = 1/4\pi$, making a corresponding change to the scattering and attenuation coefficient using similarity theory. We also replace $L_o(p, \omega) = L_i(p, -\omega)$ with a single function $L(p, \omega)$.

$$\frac{\partial}{\partial t} L(p + t\omega, \omega) = -\sigma'_t L(p, \omega) + \frac{\sigma'_s}{4\pi} \int_{S^2} L(p, \omega') d\omega' + L_e(p, \omega). \quad [15.15]$$

The key assumption of diffusion theory is that because each scattering event effectively blurs the incident illumination, high frequencies disappear from the angular radiance distribution as light propagates farther into the medium; in dense and isotropically scattering media, all directionality is eventually lost. Motivated by this observation, the radiance function is restricted to a simple two-term expansion based on spherical moments. Formally, for a function $f : S^2 \rightarrow \mathbb{R}$, the n -th moment on the unit sphere is defined as³

³ An alternative way of deriving diffusion theory involves replacing the radiance function with a low-order spherical harmonics expansion. We prefer the moment notation for its simplicity, though it should be noted that both methods are mathematically equivalent.

$$(\mu_n [f])_{i,j,k,\dots} = \int_{S^2} \underbrace{\omega_i \omega_j \omega_k \cdots}_n f(\omega) d\omega.$$

In other words, to get the i, j, k, \dots entry of the n -tensor $\mu_n [f]$, we integrate the product of f and the i, j, k, \dots components of the direction ω written in Cartesian coordinates. Note that there is some notational overlap with the angle cosines μ_k from Section 8.6; the remainder of this chapter will exclusively refer to the definition above.

The zeroth moment, for instance, gives the function's integral over the sphere, the first moment can be interpreted as a “center of mass” 3-vector, and the second moment is a positive definite 3×3 matrix. Higher order moments have many symmetries: for instance, exchanging any pair of indices leaves the value unchanged. High-order moments are useful to derive extended versions of diffusion theory that allow for more pronounced directional behavior; here, we will just focus on degrees $n \leq 1$.

The moments of the radiance function have special designations: the zeroth moment ϕ is referred to as the *fluence rate*:

$$\phi(p) = \mu_0 [L(p, \cdot)] = \int_{S^2} L(p, \omega) d\omega.$$

Note that this expression differs from the fluence function $H(p)$ in Equation (6.6), which was defined as the fluence rate on a surface boundary integrated over time.

The first moment is the *vector irradiance*:

$$E(p) = \mu_1 [L(p, \cdot)] = \int_{S^2} \omega L(p, \omega) d\omega.$$

The two-term expansion mentioned before is defined as

$$L_d(p, \omega) = \frac{1}{4\pi} \phi(p) + \frac{3}{4\pi} \omega \cdot E(p), \quad (15.16)$$

so that the moments can be exactly recovered, i.e.,

$$\mu_0 [L_d(p, \cdot)] = \phi(p) \quad \text{and} \quad \mu_1 [L_d(p, \cdot)] = E(p).$$

(Here, the “d” subscript denotes the diffusion approximation, not the direct lighting term as it did in Section 14.3.)

To derive the diffusion equation from the equation of transfer, we simply substitute the two-term radiance function L_d into Equation (15.15). The resulting expression is unfortunately not guaranteed to have a solution, but this issue can be addressed with a simple trick: by only enforcing equality of its moments, i.e., by requiring that

$$\begin{aligned} \mu_i \left[\frac{\partial}{\partial t} L_d(p + t\omega, \omega) \right] &= \mu_i \left[-\sigma_t' L_d(p, \omega) \right. \\ &\quad \left. + \frac{\sigma_s'}{4\pi} \int_{S^2} L_d(p, \omega') d\omega' + L_e(p, \omega) \right] \end{aligned} \quad (15.17)$$

for $i = 0$ and $i = 1$. Computing these moments is a fairly lengthy and mechanical exercise in trigonometric calculus that we skip here.⁴ The end result is an equation equating the zeroth moments:

$$\operatorname{div} \mathbf{E}(\mathbf{p}) = (-\sigma'_t + \sigma'_s) \phi(\mathbf{p}) = -\sigma_a \phi(\mathbf{p}) + Q_0(\mathbf{p}),$$

where $\operatorname{div} \mathbf{E}(\mathbf{p}) = \frac{\partial}{\partial x} \mathbf{E}(\mathbf{p}) + \frac{\partial}{\partial y} \mathbf{E}(\mathbf{p}) + \frac{\partial}{\partial z} \mathbf{E}(\mathbf{p})$ is the divergence operator and

$$Q_i(\mathbf{p}) = \mu_i [L_e(\mathbf{p}, \cdot)]$$

is the i -th moment of the medium emission. This equation states that the divergence of the irradiance vector field \mathbf{E} is negative in the presence of absorption (i.e., light is being removed) and positive when light is being added by Q_0 .

Another similar equation for the first moments states that the irradiance vector field \mathbf{E} , which represents the overall flow of energy, points from regions with a higher fluence rate to regions with a lower rate.

$$\frac{1}{3} \nabla \phi(\mathbf{p}) = -\sigma'_t \mathbf{E}(\mathbf{p}) + Q_1(\mathbf{p}), \quad [15.18]$$

A reasonable simplification at this point is to assume light sources in the medium emit light uniformly in all directions, in which case $Q_1(\mathbf{p}) = 0$.

The next step of the traditional derivation is to solve the above equation for \mathbf{E} and substitute it into the equation relating zeroth moments. The substitution removes $\mathbf{E}(\mathbf{p})$ and yields the *diffusion equation*, which now only involves the fluence rate $\phi(\mathbf{p})$:

$$\frac{1}{3\sigma'_t} \operatorname{div} \nabla \phi(\mathbf{p}) = \sigma_a \phi(\mathbf{p}) - Q_0(\mathbf{p}) + \frac{1}{\sigma'_t} \nabla \cdot Q_1(\mathbf{p}).$$

Assuming that $Q_1(\mathbf{p}) = 0$, the diffusion equation can be written more compactly as

$$D \nabla^2 \phi(\mathbf{p}) - \sigma_a \phi(\mathbf{p}) = -Q_0(\mathbf{p}), \quad [15.19]$$

where $D = 1/(3\sigma'_t)$ is the *classical diffusion coefficient* and ∇^2 is a shorter notation for $\operatorname{div} \nabla$, which is known as the *Laplace operator*.

With the diffusion equation at hand, we'll proceed as follows: starting from a solution for a point source that is only correct in a space where the medium infinitely extends in all directions, we will consider ways of improving the solution's accuracy in more challenging cases and introduce an approximation that can account for the effect of a refractive boundary.

We'll initially focus on a point light source that is placed below the surface to approximate the effect of incident illumination striking the surface. Later, we switch to a more accurate light source approximation and derive the beam diffusion solution to the multiple scattering component as well as a single scattering correction that is based on the classical equation of transfer.

⁴ For details, see Chapter 5 of Donner's Ph.D. dissertation (2006) or the supplemental material of Jakob et al. (2010).

15.5.3 MONPOLE SOLUTION

Consider an infinite homogeneous medium with a point light source of unit power (a *monopole*) located at its origin. The emitted radiance function L_e for this setup is given by

$$L_e(p, \omega) = \frac{1}{4\pi} \delta(p),$$

and the corresponding moments are equal to

$$Q_i(p) = \begin{cases} \delta(p), & i = 0, \\ 0, & i = 1. \end{cases}$$

The fluence rate due to this type of source has a simple analytic expression:

$$\phi_M(r) = \frac{1}{4\pi D} \frac{e^{-\sigma_{tr}r}}{r}, \quad (15.20)$$

where r is the distance from the light source. The constant $\sigma_{tr} = \sqrt{\sigma_a/D}$ is called the *effective transport coefficient*; it occurs in an exponential falloff term that accounts for absorption in the medium. Observe that $\sigma_{tr} \neq \sigma_t'$: instead, this modified attenuation coefficient additionally depends on the albedo to model the effect of multiple scattering inside the medium, hence the term *effective*.

Let's verify that this expression satisfies the diffusion equation away from the origin, where the fluence has a pole singularity. The Laplace operator ∇^2 of a function f that only depends on the radius in spherical coordinates is given by

$$\nabla^2 f = r^{-2} \frac{\partial}{\partial r} \left(r^2 \frac{\partial}{\partial r} f(r) \right).$$

Taking the inner derivative of $\phi_M(r)$ and multiplying by r^2 yields

$$r^2 \frac{\partial}{\partial r} \phi_M(r) = -r \phi_M(r) (1 + \sigma_{tr} r).$$

Taking the outer derivative, multiplying by r^{-2} , and simplifying gives

$$r^{-2} \frac{\partial}{\partial r} (-r \phi_M(r) (1 + \sigma_{tr} r)) = \sigma_{tr}^2 \phi_M(r) = \frac{\sigma_a}{D} \phi_M(r),$$

which is exactly the ratio predicted by Equation (15.19); i.e., $D\nabla^2\phi_M - \sigma_a\phi_M = 0$.

Using the identity from Equation (15.18), we can also find the irradiance vector field induced by ϕ_M , which will be useful later on:

$$\begin{aligned} E_M(p) &= -D \nabla \phi_M(p) \\ &= \left[-D \frac{\partial}{\partial r} \phi_M(r) \right] \hat{r} \\ &= \frac{1 + r\sigma_{tr}}{4\pi r^2} e^{-\sigma_{tr}r} \hat{r}, \end{aligned} \quad (15.21)$$

where \hat{r} is a unit vector pointing away from the source.

15.5.4 NON-CLASSICAL DIFFUSION

As we have seen, the monopole fluence rate ϕ_M in Equation (15.20) exactly solves the diffusion equation. Despite this, it turns out that the solution can still have significant errors compared to the original equation of transfer when the assumptions of the underlying diffusion approximation are violated.

There are two important cases: the first occurs when absorption prevents the radiance from reaching an isotropic equilibrium distribution. The second occurs close to the source, where the true radiance function is dominated by a (highly non-isotropic) spherical Dirac delta function.

A number of modified diffusion theories have been proposed that improve the accuracy in a number of different cases. An effective one is to switch to the modified monopole solution developed by Grosjean (1956) in the field of neutron transport:

$$\phi_G(r) = \frac{e^{-\sigma_t' r}}{4\pi r^2} + \tilde{\phi}_M(r). \quad [15.22]$$

The first term in this solution separates out an attenuated source term modeled using standard radiative transport—this effectively removes the portion that cannot easily be handled by diffusion. The remainder is a scaled diffusive term that accounts for light which has been scattered at least once, where the reduced albedo ρ' accounts for the energy reduction due to the extra scattering event:

$$\tilde{\phi}_M(r) = \rho' \phi_M(r). \quad [15.23]$$

This expression uses the previous monopole solution ϕ_M , though its diffusion coefficient D must be replaced with a non-classical version given by

$$D_G = \frac{2\sigma_a + \sigma_s'}{3(\sigma_a + \sigma_s')^2}. \quad [15.24]$$

Figure 15.16 illustrates the superior accuracy of Grosjean's solution in both absorption-dominated and scattering-dominated media.

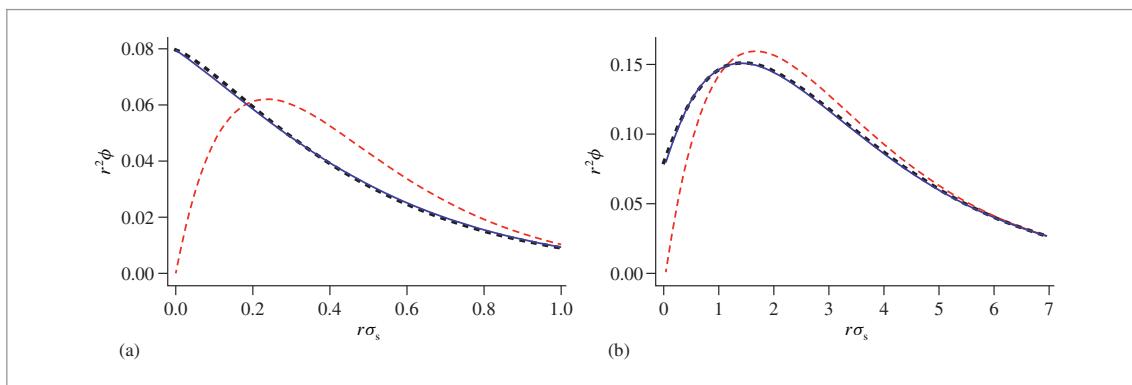


Figure 15.16: Comparison of Classical and Non-Classical Diffusion Solutions. The two plots compare the fluence rate due to a point source for (a) a low albedo ($\rho = 1/3$) and (b) a high albedo ($\rho = 0.9$). In both cases, Grosjean's non-classical monopole (blue) is a considerably better match to the exact solution (black) compared to the classical diffusion solution (red).

In the following sections, we will just focus on the non-classical diffusive part $\tilde{\phi}_M$ and ignore the attenuated source term in Equation (15.22), as it is simple to handle separately later on.

15.5.5 DIPOLE SOLUTION

To apply these results to subsurface scattering for rendering, it is clear that the solution must account for the presence of a surface. We will now switch to the simplest kind of geometric setup that satisfies this requirement: a *semi-infinite half space*, i.e., a medium that fills all space below a planar surface of infinite lateral extent. The region above is modeled as a dielectric without any kind of scattering (i.e., a vacuum).

For simplicity, we assume that the boundary is located at $z = 0$ with a normal of $\mathbf{n} = (0, 0, -1)$ so that positive values on the z axis correspond to points inside the medium. Let η denote the relative index of refraction over the boundary. We are still interested in the fluence rate due to a point light source that we (arbitrarily) place on the z -axis at position $(0, 0, z_r)$. Let's assume that $z_r > 0$, i.e., the light source is located *inside* the medium (more on this later). Due to the newly added boundary, a portion of the light traveling upward can escape from the layer and undergo no further scattering. Another portion is specularly reflected at $z = 0$; the monopole solution from Equation (15.20) accounts for neither effect and thus no longer yields accurate results.

The influence of the boundary can be approximated by the *method of images*, where a *negative* source is placed on the vacuum side of the boundary at position $(0, 0, z_v)$ with $z_v < 0$. The negative contribution of this “virtual” light source subtracts a portion of the “real” light source to account for the combined effect of internal reflections and illumination that has left the medium and can no longer scatter. The choice of the virtual depth z_v is crucial to ensure that this indeed works out—we discuss this step shortly.

This arrangement of a positive and a negative light source is known as a *dipole* (Figure 15.17). Due to the linearity of the diffusion equation, (15.19), superpositions of solutions still solve the diffusion equation; hence the dipole fluence rate at a point

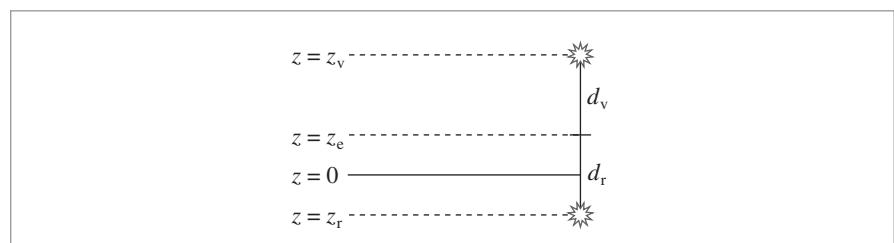


Figure 15.17: Basic Setting for the Dipole Approximation to the Solution to the Diffusion Equation. A light source with positive flux is placed at a position $z = z_r$ inside the medium, below the point where incident illumination arrives, and a second light with an equal amount of negative flux is placed at $z = z_v$ above the medium. These sources are placed so that the flux for the two of them cancels out at a height z_e above the boundary, fulfilling the linearized boundary condition. The fluence rate that results from subtracting the closed-form solution for each of them, Equation (15.25), at the medium's boundary $z = 0$ is a reasonable approximation of the fluence rate at the boundary due to subsurface scattering.

$(r, 0, 0)$ on the boundary is simply the sum of a positive and negative $\tilde{\phi}_M$ term:

$$\phi_D(r) = \tilde{\phi}_M(d_r) - \tilde{\phi}_M(d_v) \quad [15.25]$$

where $d_r = \sqrt{r^2 + z_r^2}$ and $d_v = \sqrt{r^2 + z_v^2}$ are the straight-line distances from the evaluation point to the real and virtual light source. Once more, we can find a matching vector irradiance value using the monopole solution from Equation (15.21):

$$E_D(r) = \tilde{E}_M(d_r) - \tilde{E}_M(d_v) \quad [15.26]$$

The tilde in \tilde{E}_M above indicates the use of Grosjean's modified diffusion coefficient. We will later require the z component of this expression, which is given by

$$-\mathbf{n} \cdot E_D(r) = \frac{1}{4\pi} \left[\frac{z_r(1 + d_r\sigma_{tr})}{d_r^3} e^{-\sigma_{tr}d_r} - \frac{z_v(1 + d_v\sigma_{tr})}{d_v^3} e^{-\sigma_{tr}d_v} \right]. \quad [15.27]$$

Boundary Conditions

Having defined the diffusion dipole, we must still specify how the two light sources should be placed in relation to each other so that they fulfill the appropriate boundary conditions (internal reflections, no scattering for $z < 0$). We'll assume that the real light source depth z_r is specified, and that z_v must be set to correct for the boundary.

Moulton (1990) showed that a very simple approximation is enough to get a reasonable answer. As we move from the boundary into the region filled by vacuum, it is intuitive that the fluence rate should decrease fairly quickly due to the inverse squared falloff in a region of space that does not scatter. If we model the fluence rate along the z -axis using a first-order Taylor expansion of the boundary conditions at $z = 0$, then this linear function eventually reaches 0 (and then negative values) at a *linear extrapolation depth* of $z_e < 0$.

The idea of this approach then is to mirror the real light source across the mirror plane $z = z_e$ defined by this extrapolation depth to obtain the virtual light source depth $z_v = 2z_e - z_r$; this ensures that the fluence rate of the dipole is exactly equal to 0 at $z = z_e$ above the half-space. Note that we would generally expect a point outside the medium to have a nonzero positive fluence rate due to radiance leaving the medium; here we are only interested in computing a good solution at the boundary, where the somewhat non-physical negative light source does not pose problems.

An improved variational approximation of boundary conditions for interfaces with internal Fresnel reflection was derived by Pomraning and Ganapol (1995). With their approach, the linear extrapolation depth is given by

$$z_e = -2D_G \frac{1 + 3\bar{F}_{r,2}(\eta)}{1 - 2\bar{F}_{r,1}(\eta)}, \quad [15.28]$$

where $\bar{F}_{r,1}$ and $\bar{F}_{r,2}$ are the Fresnel moments first encountered in Equation (11.8).

Radiant Exitance

At this point, we have all ingredients to compute the fluence rate due to a point source inside the surface, with corrections to account for the half-space geometry and internal

reflections. To use this solution in a light transport simulation, we'll need to know how much light actually leaves the surface.

Recall Equation (15.16), which related radiance to the fluence rate and vector irradiance. Plugging the dipole solutions into this expression yields

$$L_d(p, \omega) = \frac{1}{4\pi} \phi_D(\|p\|) + \frac{3}{4\pi} \omega \cdot E_D(\|p\|), \quad (15.29)$$

but this is only valid inside the surface. To find the amount of diffusely scattered light leaving at the boundary, we can integrate the internal radiance distribution L_d against the Fresnel transmittance and a cosine factor due to the dA^\perp term in the definition of radiance (Section 5.4).

Using linearity to split the integral into two parts that are related to the fluence rate and vector irradiance, we have

$$\begin{aligned} E_d(p) &= \int_{\mathcal{H}^2(n)} \left(1 - F_r(\eta^{-1}, \cos \theta)\right) L_d(p, \omega) \cos \theta d\omega \\ &= E_{d,\phi_D}(\|p\|) + E_{d,E_D}(\|p\|). \end{aligned} \quad (15.30)$$

When the fluence-related part is written in spherical coordinates, the integral over azimuth turns into a multiplication by 2π as no terms depend on it, and $\phi_D(r)$ can be moved out of the integral. The remaining expression reduces to a constant plus a scaled Fresnel moment (Equation (11.8)).

$$\begin{aligned} E_{d,\phi_D}(r) &= \int_0^{2\pi} \int_0^{\frac{\pi}{2}} \left(1 - F_r(\eta^{-1}, \cos \theta)\right) \frac{1}{4\pi} \phi_D(r) \cos \theta \sin \theta d\theta d\phi \\ &= \frac{1}{2} \phi_D(r) \int_0^{\frac{\pi}{2}} \left(1 - F_r(\eta^{-1}, \cos \theta)\right) \cos \theta \sin \theta d\theta \\ &= \phi_D(r) \left(\frac{1}{4} - \frac{1}{2} \bar{F}_{r,1}\right). \end{aligned} \quad (15.31)$$

For the part depending on E_D , we obtain

$$\begin{aligned} E_{d,E_D}(r) &= \int_0^{2\pi} \int_0^{\frac{\pi}{2}} \left(1 - F_r(\eta^{-1}, \cos \theta)\right) \left(\frac{3}{4\pi} \omega \cdot E_D(r)\right) \cos \theta \sin \theta d\theta d\phi \\ &= \int_0^{\frac{\pi}{2}} \left(1 - F_r(\eta^{-1}, \cos \theta)\right) \left(\frac{3 \cos \theta}{2} \mathbf{n} \cdot \mathbf{E}_D(r)\right) \cos \theta \sin \theta d\theta \\ &= \mathbf{n} \cdot \mathbf{E}_D(r) \left(\frac{1}{2} - \frac{3}{2} \bar{F}_{r,2}\right). \end{aligned} \quad (15.32)$$

In summary: after putting together all the pieces, we have a method of evaluating the radiant exitance at a position p on the boundary due to an internal source at depth z_r . So far, this depth was assumed to be a fixed parameter, but now we'll promote it to an argument of the radiant exitance we just derived, which is now written $E_d(p, z_r)$.

15.5.6 BEAM SOLUTION

At this point, we are almost ready to implement the model, though one missing piece that must still be addressed is the depth z_r of the positive point light source—this choice will clearly have an effect on the accuracy of the final solution.

The first dipole-based BSSRDF model for computer graphics proposed by Jensen et al. (2001b) placed the source at a depth of one mean free path—i.e., $z_r = 1/\sigma_t'$ —inside the medium, which is the expected distance that light will travel after entering the surface. This is a reasonable approximation, though it leads to significant errors close to the source.

With the PBD method, the point source solution is integrated over a semi-infinite interval $z_r \in [0, \infty)$ that considers all positions where light traveling along a perpendicularly incident collimated beam could scatter. The impulse response of the medium to such a spatio-directional Dirac delta function provides a more faithful description of its reflection behavior. More advanced variants of this model also allow for rays with non-perpendicular incidence. Formally, this method computes

$$E_d(p) = \int_0^{\infty} \sigma_s' e^{-\sigma_t' z_r} E_d(p, z_r) dz_r \quad (15.33)$$

The exponential models how the incident beam’s power diminishes due to extinction by the medium. Though there are a variety of high-accuracy approximations to this integral that use only a few samples, for the implementation in `pbrt`, performance is less critical since the diffusion solution is only computed once. For this reason, we use a rudimentary but simpler importance sampling scheme of the exponential term in Equation (15.33).

The function `BeamDiffusionMS()` takes the medium properties σ_s , σ_a , g , η , and a radius r and returns an average of 100 samples of the integrand.

(BSSRDF Utility Functions) ≡

```
Float BeamDiffusionMS(Float sigma_s, Float sigma_a, Float g, Float eta,
                      Float r) {
    const int nSamples = 100;
    Float Ed = 0;
    (Precompute information for dipole integrand 928)
    for (int i = 0; i < nSamples; ++i) {
        (Sample real point source depth z_r 929)
        (Evaluate dipole integrand E_d at z_r and add to Ed 929)
    }
    return Ed / nSamples;
}
```

A number of coefficients that do not depend on z_r can be precomputed outside the loop.

(Precompute information for dipole integrand) ≡

928

```
(Compute reduced scattering coefficients σ'_s, σ'_t and albedo ρ' 929)
(Compute non-classical diffusion coefficient D_G using Equation (15.24) 929)
(Compute effective transport coefficient σ_tr based on D_G 929)
(Determine linear extrapolation distance z_e using Equation (15.28) 929)
(Determine exitance scale factors using Equations (15.31) and (15.32) 929)
```

`BeamDiffusionMS()` 928

`Float` 1062

We begin by setting the reduced scattering coefficients and single scattering albedo using the principle of similarity from Section 15.5.1.

(Compute reduced scattering coefficients σ'_s , σ'_t and albedo ρ') ≡

```
Float sigmap_s = sigma_s * (1 - g);
Float sigmap_t = sigma_a + sigmap_s;
Float rhop = sigmap_s / sigmap_t;
```

928

Following this, we compute Grosjean's non-classical diffusion coefficient, Equation (15.24), and the corresponding effective transport coefficient (Section 15.5.3).

(Compute non-classical diffusion coefficient D_G using Equation (15.24)) ≡

```
Float D_g = (2 * sigma_a + sigmap_s) / (3 * sigmap_t * sigmap_t);
```

928

(Compute effective transport coefficient σ_{tr} based on D_G) ≡

```
Float sigma_tr = std::sqrt(sigma_a / D_g);
```

928

Neither the linear extrapolation depth z_e nor the scale factors from the radiant exitance computation depend on z_r , so they can also be computed. The `FresnelMoment1()` and `FresnelMoment2()` functions previously encountered in Section 11.4.1 evaluate $\bar{F}_{r,1}$ and $\bar{F}_{r,2}$.

(Determine linear extrapolation distance z_e using Equation (15.28)) ≡

```
Float fm1 = FresnelMoment1(eta), fm2 = FresnelMoment2(eta);
Float ze = -2 * D_g * (1 + 3 * fm2) / (1 - 2 * fm1);
```

928

(Determine exitance scale factors using Equations (15.31) and (15.32)) ≡

```
Float cPhi = .25f * (1 - 2 * fm1), cE = .5f * (1 - 3 * fm2);
```

928

This concludes the precomputation—all following fragments occur in the loop over samples. To select the point source depth z_r , we importance sample the homogeneous attenuation term using the same approach as in Section 15.2, setting

$$z_r = -\frac{\ln(1 - \xi_i)}{\sigma'_t} \quad (15.34)$$

for $\xi_i \in [0, 1]$. Because we are integrating a smooth 1D function, Monte Carlo isn't required. We therefore use equal-spaced positions $(i + \frac{1}{2})/N$, where $0 \leq i < N$.

(Sample real point source depth z_r) ≡

```
Float zr = -std::log(1 - (i + .5f) / nSamples) / sigmap_t;
```

928

Given z_r , we next determine the straight-line distances to the real and virtual light source, whose position is found by mirroring the real source across the linear extrapolation depth at $z = z_e$.

(Evaluate dipole integrand E_d at z_r and add to E_d) ≡

```
Float zv = -zr + 2 * ze;
Float dr = std::sqrt(r * r + zr * zr), dv = std::sqrt(r * r + zv * zv);
(Compute dipole fluence rate  $\phi_D(r)$  using Equation (15.25) 930)
(Compute dipole vector irradiance  $-\mathbf{n} \cdot \mathbf{E}_D(r)$  using Equation (15.27) 930)
(Add contribution from dipole for depth  $z_r$  to  $E_d$  930)
```

928

Float 1062

FresnelMoment1() 695

FresnelMoment2() 695

The dipole fluence rate and normal component of the irradiance were previously discussed in Equations (15.25) and (15.27).

(Compute dipole fluence rate $\phi_D(r)$ using Equation (15.25)) ≡ 929

```
Float phiD = Inv4Pi / D_g *
    (std::exp(-sigma_tr * dr) / dr - std::exp(-sigma_tr * dv) / dv);
```

(Compute dipole vector irradiance $-\mathbf{n} \cdot \mathbf{E}_D(r)$ using Equation (15.27)) ≡ 929

```
Float EDn = Inv4Pi *
    (zr * (1 + sigma_tr * dr) * std::exp(-sigma_tr * dr) / (dr*dr*dr) -
     zv * (1 + sigma_tr * dv) * std::exp(-sigma_tr * dv) / (dv*dv*dv));
```

The last fragment computes the diffuse radiant exitance E due to the dipole using Equation (15.30) and adds a scaled version of it to the running sum in Ed . In this computation, the first $rhop$ factor in the scale is needed due to the ratio of the importance sampling weight of the sampling strategy in Equation (15.34) and the σ'_s factor in Equation (15.33). The second $rhop$ factor accounts for the additional scattering event in Grosjean's non-classical monopole in Equation (15.23). Finally an empirical correction factor

$$\kappa = 1 - e^{-2\sigma_t'(d_r+z_r)}$$

corrects an overestimation that can occur when r is small and the light source is close to the surface (see Habel et al. (2013) for details).

(Add contribution from dipole for depth z_r to Ed) ≡ 929

```
Float E = phiD * cPhi + EDn * cE;
Float kappa = 1 - std::exp(-2 * sigmap_t * (dr + zr));
Ed += kappa * rhop * rhop * E;
```

15.5.7 SINGLE SCATTERING TERM

Having accounted for the diffusive multiple scattering term, it's also necessary to account for single scattering, which is not handled well by the diffusion approximation. Single scattering contributes a significant amount of energy to the scattering profile close to $r = 0$. In the absence of multiple scattering, it is fortunately simple enough to compute its contribution directly with the original equation of transfer.

Using the generalized path integral from Section 15.1.1, the radiance in the medium traveling toward position p_0 after scattering precisely once in the volume at position p_1 is given by

$$L_{ss}(p_1 \rightarrow p_0) = \hat{P}(\bar{p}_2) = \int_{p_1} L_e(p_2 \rightarrow p_1) \hat{T}(p_0, p_1, p_2) d\mu_1(p_2), \quad [15.35]$$

where \hat{T} is the generalized throughput function from Equation (15.3). We model illumination coming from a collimated beam at fixed position p_i pointing into the negative normal direction, i.e.:

$$L_e(p, \omega) = \delta(p - p_i) \delta(\omega + n). \quad [15.36]$$

Float 1062

Inv4Pi 1063

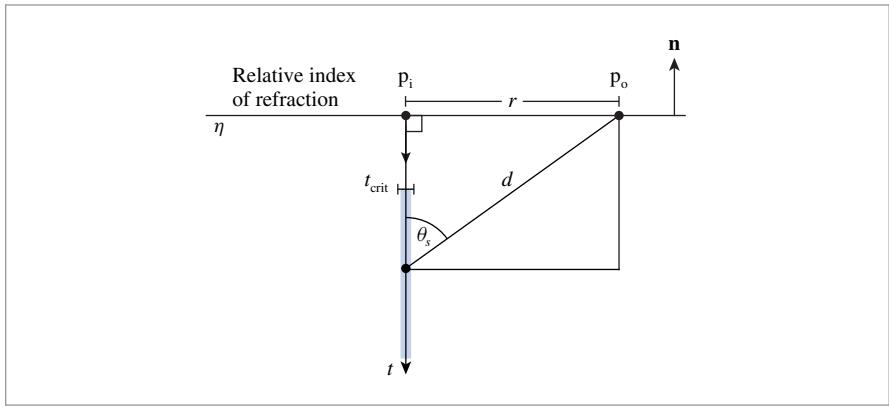


Figure 15.18: Computing the Single Scattering Portion of the BSSRDF Profile. Illumination at perpendicularly incident light is scattered once inside the material. Scattering sites before the depth t_{crit} undergo internal reflection and cannot directly contribute to single scattering. We therefore integrate the source function over the remaining depths $t \in [t_{\text{crit}}, \infty)$ to account for the full effect of single scattering.

Similar to the previous section, we are interested in the outgoing irradiance at a point p_o on the half-space boundary due to the decaying beam of light. Relevant distances and angles in this arrangement can be found using simple triangle identities (Figure 15.18). We'll briefly ignore the effect of refraction at $z = 0$ to simplify the derivation but will account for it later on in the final result.

The irradiance at p_o can be found by integrating the single-scattered radiance L_{ss} and a generalized geometry term of Equation (15.5) over the volume. Expanding the definition of L_{ss} from Equation (15.35) produces an integral over paths of length 2:

$$\begin{aligned} E_{ss}(p_o) &= \int_{\mathcal{P}_1} L_{ss}(p_1 \rightarrow p_o) \hat{G}(p_1 \leftrightarrow p_o) d\mu_1(p_1) \\ &= \int_{\mathcal{P}_2} L_e(p_2 \rightarrow p_1) \hat{T}(p_o, p_1, p_2) \hat{G}(p_1 \leftrightarrow p_o) d\mu_2(p_1, p_2). \end{aligned} \quad (15.37)$$

After expanding L_e , the spatial delta function from Equation (15.36) removes the integration over p_2 , and the directional term reduces Equation (15.37) to a 1D integral along the ray $(p_i, -n)$:

$$E_{ss}(p_o) = \int_0^\infty t^2 \hat{T}(p_o, p_i - tn, p_i) \hat{G}((p_i - tn) \leftrightarrow p_o) dt.$$

The extra t^2 term is a change of variables factor resulting from an intermediate step (not shown here), where the volumetric integral over p_1 is expressed in terms of spherical coordinates (t, ϕ, θ) . The ϕ and θ variables subsequently drop out due to the directional delta function $\delta(\omega + n)$ in L_e .

Expanding the generalized throughput \hat{T} using Equation (15.3) yields

$$\begin{aligned} E_{ss}(p_o) &= \int_0^\infty t^2 \hat{G}(p_i \leftrightarrow (p_i - t\mathbf{n})) \hat{f}(p_i \rightarrow (p_i - t\mathbf{n}) \rightarrow p_o) \\ &\quad \times \hat{G}((p_i - t\mathbf{n}) \leftrightarrow p_o) dt \end{aligned} \quad [15.38]$$

The definition of the generalized scattering function from Equation (15.4) and the assumption that the phase function only depends on the cosine of the scattering angle $\cos \theta_s$ imply that \hat{f} can be written as

$$\hat{f}(p_i \rightarrow (p_i - t\mathbf{n}) \rightarrow p_o) = \sigma_s p(-\cos \theta_s). \quad [15.39]$$

We will use the Henyey–Greenstein phase function model for p . The angle cosine $\cos \theta_s$ can be found from the triangle edge lengths (Figure 15.18):

$$\cos \theta_s = \frac{t}{d}. \quad [15.40]$$

The hypotenuse d is the distance from the scattering location $p_i - t\mathbf{n}$ to the exit point p_o and is given by the Pythagorean theorem:

$$d = \sqrt{r^2 + t^2}, \quad [15.41]$$

where $r = \|p_i - p_o\|$.

Due to the assumptions of a homogeneous medium without internal occluders, the first geometric term in Equation (15.38) takes on a simple form:

$$\hat{G}(p_i \leftrightarrow (p_i - t\mathbf{n})) = \frac{e^{-\sigma_t t}}{t^2}. \quad [15.42]$$

Note that this equation uses the original attenuation coefficient σ_t rather than the reduced version σ'_t from Section 15.5.1. In contrast to diffusion theory, the equation of transfer easily accounts for anisotropy; hence there is no longer a need for this approximation.

For the second geometry term, we must include a cosine factor that accounts for the angle that the connecting ray segment makes with the surface normal \mathbf{n} when intersecting the boundary at p_o :

$$\hat{G}((p_i - t\mathbf{n}) \leftrightarrow p_o) = \frac{e^{-\sigma_t d}}{d^2} |\cos \theta_o|. \quad [15.43]$$

The two cosine terms $\cos \theta_o$ and $\cos \theta_s$ are equal except for a difference in sign, which can be seen from the triangle geometry in Figure 15.18.

Plugging Equations (15.42), (15.43), and (15.39) into Equation (15.38) yields the following expression for the irradiance due to single scattering:

$$E_{ss}(p_o) = \int_0^\infty \frac{\sigma_s e^{-\sigma_t(t+d)}}{d^2} p(\cos \theta_s) |\cos \theta_o| dt.$$

At this point, we'll reintroduce the effect of the refractive boundary by adding a Fresnel transmission factor $(1 - F_r(\eta, \cos \theta_o))$. Light that is internally reflected by the boundary

is already included in the diffusion solution and should be excluded from the single-scattering profile. As before in Section 15.5.6, responsibility for the first refraction at p_i is delegated to the Material's BSDF and thus there is just a single Fresnel transmission term,

$$E_{ss, F_r}(p_o) = \int_0^\infty \frac{\sigma_s e^{-\sigma_t(t+d)}}{d^2} p(\cos \theta_s) (1 - F_r(\eta, \cos \theta_o)) |\cos \theta_o| dt. \quad (15.44)$$

Before we begin with the implementation of this integral, there is one more effect that should be considered: when the relative index of refraction η over the boundary is greater than 1, no illumination can directly leave the material at angles below the critical angle θ_{crit} due to total internal reflection, where

$$\theta_{crit} = \sin^{-1} \frac{1}{\eta}.$$

To avoid unnecessary computations for scattering locations that cannot possibly contribute, we restrict the integral to the range satisfying

$$\cos \theta_o < -\cos \theta_{crit} = -\sqrt{1 - \frac{1}{\eta^2}}.$$

Combining this with Equation (15.41) and solving for t yields

$$t > t_{crit} = r \sqrt{\eta^2 - 1}. \quad (15.45)$$

As in Section 15.5.6, we compute this integral by uniformly sampling distances t_i according to exponential distribution (which now starts at depth t_{crit}) and set

$$t_i = t_{crit} - \frac{\ln(1 - \xi_i)}{\sigma_t} \quad (15.46)$$

for evenly spaced $\xi_i \in [0, 1]$. The associated PDF is

$$p_t(t) = \sigma_t e^{-\sigma_t(t-t_{crit})},$$

and dividing the integrand in Equation (15.44) by this PDF results in a throughput weight of

$$\beta = \frac{\rho e^{-\sigma_t(t_{crit}+d)}}{d^2} p(-\cos \theta_o) (1 - F_r(\eta, -\cos \theta_o)) \cos \theta_o, \quad (15.47)$$

where ρ was previously defined as σ_s/σ_t .

The PBD single-scattering profile computation is implemented in `BeamDiffusionSS()`, which takes the medium scattering properties and a radius r as input. One hundred samples are used for the integral estimate, as in `BeamDiffusionMS()`.

`BeamDiffusionMS()` 928

`BeamDiffusionSS()` 934

BSDF 572

Material 577

```
(BSSRDF Utility Functions) +≡
  Float BeamDiffusionSS(Float sigma_s, Float sigma_a, Float g, Float eta,
                        Float r) {
    (Compute material parameters and minimum t below the critical angle 934)
    Float Ess = 0;
    const int nSamples = 100;
    for (int i = 0; i < nSamples; ++i) {
      (Evaluate single-scattering integrand and add to Ess 934)
    }
    return Ess / nSamples;
}
```

The function begins by precomputing derived material parameters and setting t_{crit} to the minimal distance below the critical angle as specified by Equation (15.45).

(Compute material parameters and minimum t below the critical angle) ≡ 934

```
  Float sigma_t = sigma_a + sigma_s, rho = sigma_s / sigma_t;
  Float tCrit = r * std::sqrt(eta * eta - 1);
```

The loop body generates distances t_i according to the sampling scheme in Equation (15.46).

(Evaluate single-scattering integrand and add to Ess) ≡ 934

```
  Float ti = tCrit - std::log(1 - (i + .5f) / nSamples) / sigma_t;
  (Determine length d of connecting segment and cos θ₀ 934)
  (Add contribution of single scattering at depth t 934)
```

Next, the function computes the length of the connection segment using Equation (15.41). $\cos \theta_0$ is given by Equation (15.40) except for a sign flip that is needed since the half-space normal \mathbf{n} points away from the medium.

(Determine length d of connecting segment and cos θ₀) ≡ 934

```
  Float d = std::sqrt(r * r + ti * ti);
  Float cosTheta0 = ti / d;
```

The last loop statement accumulates the throughput weight β from Equation (15.47) into the running sum Ess .

(Add contribution of single scattering at depth t) ≡ 934

```
  Ess += rho * std::exp(-sigma_t * (d + tCrit)) / (d * d) *
    PhaseHG(cosTheta0, g) * (1 - FrDielectric(-cosTheta0, 1, eta)) *
    std::abs(cosTheta0);
```

15.5.8 FILLING THE BSSRDTABLE

With the definitions of `BeamDiffusionMS()` and `BeamDiffusionSS()` at hand, we'll now implement the function `ComputeBeamDiffusionBSSRDF()` that uses these functions to fill the `BSSRDTTable` of a `TabulatedBSSRDF` with profile data.

`ComputeBeamDiffusionBSSRDF()` takes the medium's anisotropy parameter g and relative index of refraction η as input and initializes a `BSSRDTTable` that stores these quantities as

`BeamDiffusionMS()` 928
`BeamDiffusionSS()` 934
`BSSRDTTable` 697
`ComputeBeamDiffusionBSSRDF()` 935
`Float` 1062
`FrDielectric()` 519
`PhaseHG()` 681
`TabulatedBSSRDF` 696

a function of radius and albedo. This function is in turn invoked by the initialization routines of the `SubsurfaceMaterial` and `KdSubsurfaceMaterial` using a default `BSSRDFTable` with 100 albedo samples and 64 radius samples.

```
<BSSRDF Utility Functions> +≡
void ComputeBeamDiffusionBSSRDF(Float g, Float eta, BSSRDFTable *t) {
    (Choose radius values of the diffusion profile discretization 935)
    (Choose albedo values of the diffusion profile discretization 936)
    ParallelFor(
        [&](int i) {
            (Compute the diffusion profile for the i-th albedo sample 936)
        }, t->nRhoSamples);
}
```

Both single- and multiple-scattering components of the profile function $S_t(r)$ are characterized by an exponential decay with increasing radius r . By placing many samples in high-valued regions and comparably fewer in low-valued regions, we can obtain a better spline approximation of the true profile. The following fragment places the first radius sample at 0 and distributes the remaining samples at exponentially increasing distances. (Note that radius values are unitless and independent of the actual medium density σ_t . These unitless optical radii were previously introduced in Section 11.4.2.)

```
(Choose radius values of the diffusion profile discretization) ≡ 935
t->radiusSamples[0] = 0;
t->radiusSamples[1] = 2.5e-3f;
for (int i = 2; i < t->nRadiusSamples; ++i)
    t->radiusSamples[i] = t->radiusSamples[i - 1] * 1.2f;
```

Next, we need to decide on the locations of N albedo samples ρ_i on the interval $[0, 1]$. Some precautions must be taken here as well, since the material's scattering behavior has a highly nonlinear dependence on the ρ parameter.

Consider the example of a medium with a single scattering albedo of $\rho = 0.8$: this produces a surprisingly absorptive BSSRDF with an effective albedo ρ_{eff} of less than 0.15! The reason for this behavior is that most incident illumination is scattered many times before it finally leaves the half space; each scattering event incurs an energy reduction by ρ , leading to this striking nonlinearity. The left side of Figure 15.19 plots ρ_{eff} as a function of ρ , which shows that most of the interesting behavior is crammed into a small region near $\rho \approx 1$.

Clearly, uniform sample placement $\rho_i = i/(N - 1)$ will not capture this behavior in a satisfactory way. Instead, we use a heuristic that approximately inverts the nonlinear mapping between ρ and ρ_{eff} :

`BSSRDFTable` 697
`BSSRDFTable::nRadiusSamples` 698
`BSSRDFTable::nRhoSamples` 698
`BSSRDFTable::radiusSamples` 698
`Float` 1062
`KdSubsurfaceMaterial` 701
`ParallelFor()` 1088
`SubsurfaceMaterial` 700

$$\rho_i = \frac{1 - e^{-8i/(N-1)}}{1 - e^{-8}}, \quad [15.48]$$

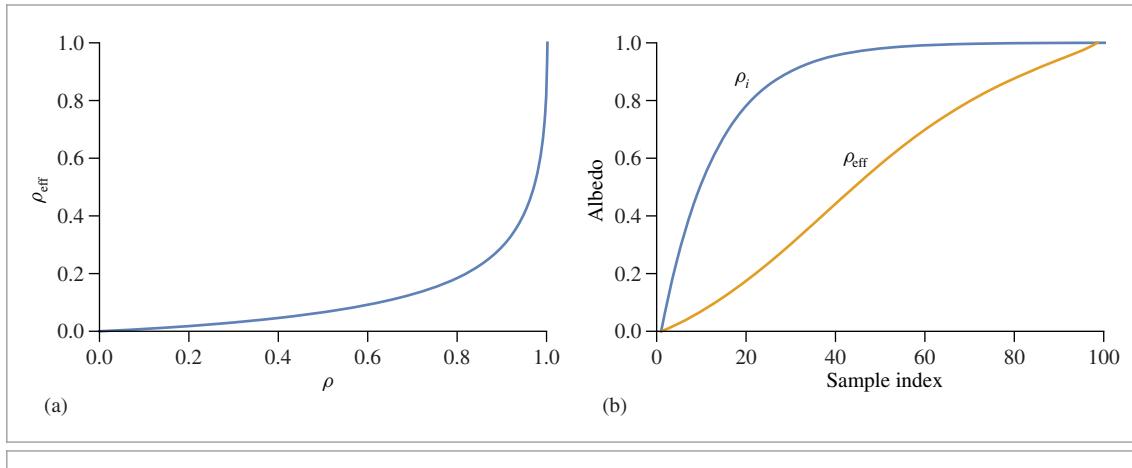


Figure 15.19: (a) The relationship between the single scattering albedo ρ and the effective albedo ρ_{eff} is highly nonlinear. (b) We use the reparameterization in Equation (15.48) to achieve a more effective sample placement with an approximately uniform sample spacing in effective albedo space.

(Choose albedo values of the diffusion profile discretization) \equiv

935

```
for (int i = 0; i < t->nRhoSamples; ++i)
    t->rhoSamples[i] =
        (1 - std::exp(-8 * i / (Float)(t->nRhoSamples - 1))) /
        (1 - std::exp(-8));
```

This results in a more perceptually uniform placement of the albedo samples, which can be seen on the right side of Figure 15.19: while not a perfect straight line, the effective albedo associated with increasing sample indices i is now reasonably close to linear.

The loop body then iterates over albedo samples and computes a diffusion profile and associated effective albedo for each one.

(Compute the diffusion profile for the i th albedo sample) \equiv

935

```
(Compute scattering profile for chosen albedo  $\rho$  937)
(Compute effective albedo  $\rho_{\text{eff}}$  and CDF for importance sampling 937)
```

The first fragment interprets the scattering profile $S_r(r)$ as a distribution in polar coordinates (r, ϕ) . The marginal distribution over the radius parameter is then given by $2\pi r S_r(r)$, where $S_r(r) = S_{r,\text{ss}}(r) + S_{r,\text{ms}}(r)$ is the sum of single and multiple scattering profiles.

The reason for tabulating the marginal distribution rather than the profile itself is to facilitate sample generation: in this way, `BSSRDFTable::profile` can be treated as parameterized sequence of 1D distributions that can be sampled using existing tools like `SampleCatmullRom2D()` (as is done in `TabulatedBSSRDF::Sample_Sr()`). However, we must be careful to cancel the extra factor of $2\pi r$ during normal profile evaluations—this was implemented in the fragment *(Cancel marginal PDF factor from tabulated BSSRDF profile)* in Section 11.4.2.

`BSSRDFTable::nRhoSamples` 698

`BSSRDFTable::profile` 698

`BSSRDFTable::rhoSamples` 698

`Float` 1062

`SampleCatmullRom2D()` 824

`TabulatedBSSRDF::Sample_Sr()`

914

```
(Compute scattering profile for chosen albedo ρ) ≡ 936
    for (int j = 0; j < t->nRadiusSamples; ++j) {
        float rho = t->rhoSamples[i], r = t->radiusSamples[j];
        t->profile[i * t->nRadiusSamples + j] = 2 * Pi * r *
            (BeamDiffusionSS(rho, 1 - rho, g, eta, r) +
            BeamDiffusionMS(rho, 1 - rho, g, eta, r));
    }
```

Integrating the (unnormalized) marginal PDF over the interval $r \in [0, \infty)$ yields the effective albedo, which was defined in Equation (11.11). In practice, we limit the integral to a finite interval $[0, r_{\max}]$ —this is not an issue, as $S_r(r)$ is negligible for $r > r_{\max}$. The `IntegrateCatmullRom()` function computes this integral in addition to an auxiliary CDF for importance sampling.

```
(Compute effective albedo ρeff and CDF for importance sampling) ≡ 936
    t->rhoEff[i] =
        IntegrateCatmullRom(t->nRadiusSamples, t->radiusSamples.get(),
            &t->profile[i * t->nRadiusSamples],
            &t->profileCDF[i * t->nRadiusSamples]);
```

`pbrt` uses the inversion method to importance sample piecewise cubic spline functions. This entails first choosing a spline segment according to a discrete probability mass function and then picking a position inside the segment. The `IntegrateCatmullRom()` function therefore computes a cumulative distribution function that is useful for implementing this sampling operation in an efficient way.

```
(Spline Interpolation Definitions) +≡
    float IntegrateCatmullRom(int n, const float *x, const float *values,
        float *cdf) {
        float sum = 0;
        cdf[0] = 0;
        for (int i = 0; i < n - 1; ++i) {
            (Look up xi and function values of spline segment i 823)
            (Approximate derivatives using finite differences 823)
            (Keep a running sum and build a cumulative distribution function 938)
        }
        return sum;
    }
```

BeamDiffusionMS() 928
 BeamDiffusionSS() 934
 BSSRDTTable::nRadiusSamples 698
 BSSRDTTable::profile 698
 BSSRDTTable::profileCDF 914
 BSSRDTTable::radiusSamples 698
 BSSRDTTable::rhoEff 698
 BSSRDTTable::rhoSamples 698
 float 1062
 IntegrateCatmullRom() 937
 pi 1063
 TabulatedBSSRDF::rho 697

The loop iterates over each spline segment and computes the definite integral of the associated cubic spline interpolant, Equation (8.27), over the associated interval. This definite integral has a simple analytic solution:

$$\int_{x_i}^{x_{i+1}} p_i(x) dx = (x_{i+1} - x_i) \left[\frac{f(x_i) + f(x_{i+1})}{2} + \frac{f'(x_i) - f'(x_{i+1})}{12} \right],$$

where p_i is the spline interpolant defined on the interval $[x_i, x_{i+1}]$, the values $f(x_i)$ and $f(x_{i+1})$ are function evaluations at the endpoints, and $f'(x_i)$ and $f'(x_{i+1})$ are derivative estimates.

The first two fragments in the loop body were previously discussed—they look up the function values (denoted by f_0 and f_1) and derivative estimates (denoted by d_0 and d_1) in the current interval and initialize `width` with the interval length.

We can now compute the definite integral and add it to a running sum. Partial sums are written into the `cdf` array.

(Keep a running sum and build a cumulative distribution function) ≡ 937

```
sum += ((d0 - d1) * (1.f / 12.f) + (f0 + f1) * .5f) * width;
cdf[i + 1] = sum;
```

15.5.9 SETTING SCATTERING PROPERTIES

It is remarkably unintuitive to set values of the absorption and scattering coefficients σ_a and σ_s to achieve a desired visual result. If measured values of these parameters aren't available (e.g., from values from the `GetMediumScatteringProperties()` utility function of Section 11.4.3), then the task for an artist trying to render subsurface scattering can be difficult.

In this section, we define a convenience function `SubsurfaceFromDiffuse()` used by the `KdSubsurfaceMaterial`, which solves an inverse problem using information stored in the `BSSRDFTable` to derive the medium's scattering properties. The function takes considerably more intuitive parameters as input: in addition to the `BSSRDFTable`, it requires an effective albedo and the average distance light travels in the medium before scattering (the mean free path length).

The medium can be made more transparent by increasing the mean free path length or denser by decreasing it. The amount of multiple scattering can be controlled by how close the effective albedo is to 1. The remaining medium properties (index of refraction and scattering anisotropy) are assumed to be fixed.

We perform the inversion separately for each wavelength using `InvertCatmullRom()` to map from effective albedo to a single scattering albedo ρ . With ρ known, the desired coefficients are given by $\sigma_s = \rho/\sigma_t^{-1}$ and $\sigma_a = (1 - \rho)/\sigma_t^{-1}$, where σ_t is the reciprocal of the mean free path length.

(BSSRDF Utility Functions) +≡

```
void SubsurfaceFromDiffuse(const BSSRDFTable &t, const Spectrum &rhoEff,
    const Spectrum &mfp, Spectrum *sigma_a, Spectrum *sigma_s) {
    for (int c = 0; c < Spectrum::nSamples; ++c) {
        Float rho = InvertCatmullRom(t.nRhoSamples, t.rhoSamples.get(),
            t.rhoEff.get(), rhoEff[c]);
        (*sigma_s)[c] = rho / mfp[c];
        (*sigma_a)[c] = (1 - rho) / mfp[c];
    }
}
```

BSSRDFTable 697
BSSRDFTable::nRhoSamples 698
BSSRDFTable::rhoEff 698
BSSRDFTable::rhoSamples 698
CoefficientSpectrum::nSamples 318
Float 1062
GetMediumScatteringProperties() 702
InvertCatmullRom() 939
KdSubsurfaceMaterial 701
SampleCatmullRom() 823
Spectrum 315
SubsurfaceFromDiffuse() 938

`InvertCatmullRom()` is very similar to the already defined `SampleCatmullRom()` function except that it directly inverts the spline function and not its definite integral—it otherwise applies the same Newton-Bisection algorithm (and reuses a number of code fragments). We therefore won't include its implementation here. Note that using this approach

requires that the underlying function be either monotonically increasing or monotonically decreasing; in the case of `SubsurfaceFromDiffuse()`, the function is monotonically increasing.

(Spline Interpolation Declarations) ≡

```
Float InvertCatmullRom(int n, const Float *x, const Float *values,
                      Float u);
```

FURTHER READING

Lommel (1889) was apparently the first to derive the equation of transfer. Not only did he derive the equation of transfer, but he also solved it in some simplified cases in order to estimate reflection functions from real-world surfaces (including marble and paper) and compared his solutions to measured reflectance data from these surfaces.

Seemingly unaware of Lommel’s work, Schuster (1905) was the next researcher in radiative transfer to consider the effect of multiple scattering. He used the term *self-illumination* to describe the fact that each part of the medium is illuminated by every other part of the medium, and he derived differential equations that described reflection from a slab along the normal direction assuming the presence of isotropic scattering. The conceptual framework that he developed remains essentially unchanged in the field of radiative transfer.

Soon thereafter, Schwarzschild (1906) introduced the concept of radiative equilibrium, and Jackson (1910) expressed Schuster’s equation in integral form, also noting that “the obvious physical mode of solution is Liouville’s method of successive substitutions” (i.e., a Neumann series solution). Finally, King (1913) completed the rediscovery of the equation of transfer by expressing it in the general integral form. Yanovitskij (1997) traced the origin of the integral equation of transfer to Chvolson (1890), but we have been unable to find a copy of this paper.

Books by Chandrasekhar (1960), Preisendorfer (1965, 1976), and van de Hulst (1980) cover volume light transport in depth.

Blinn (1982b) first used basic volume scattering algorithms for computer graphics. The equation of transfer was first introduced to graphics by Kajiya and Von Herzen (1984). Rushmeier (1988) was the first to compute solutions of it in a general setting. Arvo (1993) first made the essential connections between previous formalizations of light transport in graphics and the equation of transfer and radiative transfer in general. Pauly, Kollig, and Keller (2000) derived the generalization of the path integral form of the light transport equation for the volume scattering case.

See also the “Further Reading” section of Chapter 11 for additional references to previous work on light scattering in participating media.

Building Blocks

Float 1062

The paper by Raab et al. (2006) introduced many important sampling building-blocks for rendering participating media to graphics, including the delta-tracking algorithm for inhomogeneous media by Woodcock et al. (1965). (pbrt uses this algorithm in

`GridDensityMedium::Sample()`) More recently, Novák et al. (2014) derived *ratio tracking* and *residual ratio tracking*, which provide unbiased estimates of the transmittance function in inhomogeneous media with considerably lower variance than delta tracking; the simpler ratio tracking algorithm is used in `GridDensityMedium::Tr()`.

For media with substantial variation in density, delta tracking can be quite inefficient—many small steps must be taken to get through the optically thin sections. Danskin and Hanrahan (1992) presented a technique for efficient volume ray marching using a hierarchical data structure. Another way of addressing this issue was presented by Szirmay-Kalos et al. (2011), who used a grid to partition scattering volumes in cells and applied delta tracking using the maximum density cells as the ray passed through them. Yue et al. (2010) applied a similar approach but used a kd-tree, which was better able to adapt to spatially varying densities. In follow-on work, they derive an approach to estimate the efficiency of spatial partitionings and use it to construct them more effectively (Yue et al. 2011).

Kulla and Fajardo (2012) noted that techniques based on sampling beam transmittance ignore another important factor: spatial variation in the scattering coefficient. They developed a method based on computing a tabularized 1D sampling distribution for each ray passing through participating media based on the product of beam transmittance and scattering coefficient at a number of points along it. They then draw samples from this distribution, showing good results.

Accounting for illumination from emissive media is important for many effects, including fire and explosions. See Villemain and Hery (2013) for algorithms for sampling illumination from these sorts of emitters.

Researchers have recently had success in deriving closed-form expressions that describe scattering along unoccluded ray segments in participating media; these approaches can be substantially more efficient than integrating over a series of point samples. See Sun et al. (2005), Pegoraro and Parker (2009), and Pegoraro et al. (2009, 2010, 2011) for examples of such methods. (Remarkably, Pegoraro and collaborators' work provides a closed-form expression for scattering from a point light source along a ray passing through homogeneous participating media with anisotropic phase functions.)

Light Transport Algorithms

Rushmeier and Torrance (1987) used finite-element methods for rendering participating media. Other early work in volume scattering for computer graphics includes work by Max (1986), Nishita, Miyawaki, and Nakamae (1987), Bhate and Tokuta's approach based on spherical harmonics (Bhate and Tokuta 1992), and Blasi et al.'s two-pass Monte Carlo algorithm, where the first pass shoots energy from the lights and stores it in a grid and the second pass does final rendering using the grid to estimate illumination at points in the scene (Blasi, Saëc, and Schlick 1993). Glassner (1995) provided a thorough overview of this topic and previous applications of it in graphics, and Max's survey article (Max 1995) also covers early work well. See Cerezo et al. (2005) for an extensive survey of approaches to rendering participating media up through 2005.

More recently, Szirmay-Kalos et al. (2005) precomputed interactions between sample points in the medium in order to more quickly compute multiple scattering. Pegoraro et al. (2008b) developed an interesting approach for improving Monte Carlo rendering

`GridDensityMedium::Sample()`
896

`GridDensityMedium::Tr()` 898

of participating media by using information from previous samples to guide future sampling.

Georgiev et al. (2013) made the observation that incremental path sampling can generate particularly bad paths in participating media. They proposed new multi-vertex sampling methods that better account for all of the relevant terms in the equation of transfer.

Sampling direct illumination from lights at points inside media surrounded by a primitive is challenging; traditional direct lighting algorithms aren't applicable at points inside the medium, as refraction through the primitive's boundary will divert the shadow ray's path. Walter et al. (2009) considered this problem and developed algorithms to efficiently find paths to lights accounting for this refraction.

The visual appearance of high albedo objects like clouds is striking, but many bounces may be necessary for good results. Wrenninge et al. (2013) described an approximation where after the first few bounces, the scattering coefficient, attenuation coefficient for shadow rays, and the eccentricity of the phase function are all progressively reduced. (This approach draws from ideas behind the principle of similarity from Section 15.5.1.)

All of the bidirectional light transport algorithms that will be introduced in Chapter 16 can be extended to handle participating media; most of our implementations include these extensions. See the “Further Reading” section in that chapter for references to previous work on these topics. See also Jarosz’s thesis (2008), which has extensive background on this topic (and includes a number of important contributions).

Subsurface Scattering

Subsurface scattering was first introduced to graphics by Hanrahan and Krueger (1993), although their approach did not attempt to simulate light that entered the object at points other than at the point being shaded. Dorsey et al. (1999) applied photon maps to simulating subsurface scattering that did include this effect, and Pharr and Hanrahan (2000) introduced an approach based on computing BSSRDFs for arbitrary scattering media with an integral over the medium’s depth.

Kajiya and Von Herzen (1984) first introduced the diffusion approximation to graphics, though Stam (1995) was the first to clearly identify many of its advantages for rendering. See Ishimaru’s book (1978) or Donner’s thesis (2006) for the derivation of the diffusion approximation and Wyman et al. (1989) for the introduction of the principle of similarity. More recently, Zhao et al. (2014) further investigated the similarity relations, derived higher order relations, and showed their application to rendering.

The dipole approximation for subsurface scattering was developed by Farrell et al. (1992). It was introduced to computer graphics by Jensen et al. (2001b). Jensen and Buhler (2002) developed an efficient hierarchical integration approach based on precomputing incident irradiance at a set of points on the primitive’s surface. The dipole approximation saw early application to production rendering via a scan-line implementation (Hery 2003).

Contini et al. (1997) generalized the dipole approach to *multipoles* to more accurately model finite scattering slabs. This approach was applied to subsurface scattering by Donner and Jensen (2005). However, even the multipole approach doesn’t handle all types of scattering media well; the assumptions of homogeneous media and relatively high

albedos are too restrictive for many interesting objects. Li et al. (2005) developed a hybrid approach that handles the first few bounces of light with Monte Carlo path tracing but then switches to a dipole approximation. Tong et al. (2005) developed a method to capture and render materials with small deviations from an overall homogeneous appearance. Haber et al. (2005b), and Wang et al. (2008b) further generalized the media supported, solving the diffusion equation on a grid of sample points. Fattal (2009) applied the discrete ordinates method, addressing a number of shortcomings of the direct application of that technique. Arbree et al. (2011) developed a finite element method to solve the diffusion equation on a tetrahedral mesh in a way that is more numerically robust than prior grid-based methods.

The photon beam diffusion approach implemented in Section 15.5 is based on the approach developed by Habel et al. (2013). It builds on the quantized diffusion model developed by d'Eon and Irving (2011), who introduced the Grosjean monopole (Grosjean 1956), the approach of Pomraning and Ganapol (1995) for computing the dipole depth in Equation (15.28), and the computation of the radiant exitance using the approach proposed by Kienle and Patterson (1997).

Frisvad et al. (2014) developed an alternative diffusion technique to model subsurface scattering due to an incident beam of light; in contrast to photon beam diffusion, which integrates isotropic sources along continuous beams, their method builds on a discrete arrangement of anisotropic (i.e., $Q_1 \neq 0$) monopole solutions.

While much effort has gone into more accurate diffusion profiles, Christensen and Burley (2015) showed that a simple exponential approximation to these profiles fits them extremely well and is quite efficient to evaluate.

Donner et al. (2009) computed BSSRDFs with Monte Carlo simulation for a variety of scattering properties (phase function, scattering coefficients, etc.) and fit the resulting data to a low-dimensional model. This model accurately accounts for the directional variation of scattered light and the properties of medium-albedo media.

Rendering realistic human skin is a challenging problem; this problem has driven the development of a number of new methods for rendering subsurface scattering after the initial dipole work as issues of modeling the layers of skin and computing more accurate simulations of scattering between layers have been addressed. For a good overview of these issues, see Igarashi et al.'s (2007) survey on the scattering mechanisms inside skin and approaches for measuring and rendering skin. Notable research in this area includes papers by Donner and Jensen (2006), d'Eon et al. (2007), Ghosh et al. (2008), and Donner et al. (2008). Donner's thesis includes a discussion of the importance of accurate spectral representations for high-quality skin rendering (Donner 2006, Section 8.5).

The algorithm implemented in Section 15.4.1 to find sample points for incident illumination for BSSRDFs was developed by King et al. (2013).

Other Topics

One key application of volume scattering algorithms in computer graphics has been simulating atmospheric scattering. Work in this area includes early papers by Klassen (1987) and Preetham et al. (1999), who introduced a physically rigorous and computationally efficient atmospheric and sky-lighting model. Haber et al. (2005a) described a model for

twilight, and Hošek and Wilkie (2012, 2013) developed a comprehensive model for sky-and sun-light.

There are a number of applications of visualizing volumetric data sets for medical and engineering applications. This area is called *volume rendering*. In many of these applications, radiometric accuracy is substantially less important than developing techniques that help make structure in the data apparent (e.g., where the bones are in CT scan data). Early papers in this area include those by Levoy (1988, 1990a, 1990b) and Drebin, Carpenter, and Hanrahan (1988).

Moon et al. (2007) made the important observation that some of the assumptions underlying the use of the equation of transfer—that the scattering particles in the medium aren’t too close together so that scattering events can be considered to be statistically independent—aren’t in fact true for interesting scenes that include small crystals, ice, or piles of many small glass objects. They developed a new light transport algorithm for these types of *discrete random media* based on composing precomputed scattering solutions.

Jakob et al. (2010) derived a generalized transfer equation that describes scattering by distributions of oriented particles. They proposed a *microflake* scattering model as a specific example of a particle distribution (where a microflake is the volumetric analog of a microfacet on a surface) and showed a number of ways of solving this equation based on Monte Carlo, finite elements, and a dipole model. More recently, Heitz et al. (2015) derived a generalized microflake distribution, which is considerably more efficient to sample and evaluate. Their model quantifies the local scattering properties using projected areas observed from different directions, which adds a well-defined notion of volumetric level of detail.

The equation of transfer assumes that the index of refraction of a medium will only change at discrete boundaries, though many actual media have continuously varying indices of refraction. Ament et al. (2014) derived a variant of the equation of transfer that allows for this case and applied photon mapping to render images with it.

EXERCISES

- ② 15.1 With optically dense inhomogeneous volume regions, `GridDensityMedium::Tr()` may spend a lot of time finding the attenuation between lights and intersection points. One approach to reducing this expense is to take advantage of the facts that the amount of attenuation for nearby rays is generally smoothly varying and that the rays to a point or directional light source can be parameterized over a straightforward 2D domain. Given these conditions, it’s possible to use precomputed approximations to the attenuation.

For example, Kajiya and Von Herzen (1984) computed the attenuation to a directional light source at a grid of points in 3D space and then found attenuation at any particular point by interpolating among nearby grid samples. A more memory-efficient approach was developed by Lokovic and Veach (2000) in the form of deep shadow maps, based on a clever compression technique that takes advantage of the smoothness of the attenuation. Implement one of

these approaches in `pbrt`, and measure how much it speeds up rendering with the `VolPathIntegrator`. Under what sorts of situations do approaches like these result in noticeable image errors?

- ➊ 15.2 Another effective method for speeding up `GridDensityMedium::Tr()` is to use Russian roulette: if the accumulated transmittance Tr goes below some threshold, randomly terminate it and return 0 transmittance; otherwise, scale it based on 1 over the survival probability. Modify `pbrt` to optionally use this approach, and measure the change in Monte Carlo efficiency. How does varying the termination threshold affect your results?
- ➋ 15.3 Read the papers by Yue et al. (2010, 2011) on improving delta-tracking’s efficiency by decomposing inhomogeneous media using a spatial data structure and then applying delta tracking separately in each region of space. Apply their approach to the `GridDensityMedium`, and measure the change in efficiency compared to the current implementation.
- ➌ 15.4 The current sampling algorithm in the `GridDensityMedium` is based purely on sampling based on the accumulated attenuation. While this more effective than sampling uniformly, it misses the factor that it’s desirable to sample scattering events at points where the scattering coefficient is relatively large as well, as these points contribute more to the overall result. Kulla and Fajardo (2012) describe an approach based on sampling the medium at a number of points along each ray and computing a PDF for the product of the transmittance and the scattering coefficient. Sampling from this distribution gives much better results than sampling based on the transmittance alone.

Implement Kulla and Fajardo’s technique in `pbrt`, and compare the Monte Carlo efficiency of their method to the method currently implemented in `GridDensityMedium`. Are there scenes where their approach is less effective?

- ➍ 15.5 As described in Section 15.3.1, the current `VolPathIntegrator` implementation will spend unnecessary effort computing ray-primitive intersections in scenes with optically dense scattering media: closer medium interactions will often be sampled than the surface intersections. Modify the system so that medium interactions are sampled before ray-primitive intersections are tested. Reduce the ray’s `tMax` extent when a medium interaction is sampled before performing primitive intersections. Measure the change in performance for scenes with both optically thin and optically thick participating media. (Use a fairly geometrically complex scene so that the cost of ray-primitive intersections isn’t negligible.) If your results show that the most efficient approach varies depending on the medium scattering properties, implement an approach to automatically choose between the two strategies at run time based on the medium’s characteristics.
- ➎ 15.6 The `Medium` abstraction currently doesn’t make it possible to represent emissive media, and the volume-aware integrators don’t account for volumetric emission. Modify the system so that emission from a 3D volume can be described, and update one or more `Integrator` implementations to account for emissive media in their lighting calculations. For the code related to sampling incident

`GridDensityMedium` 690

`GridDensityMedium::Tr()` 898

`Integrator` 25

`VolPathIntegrator` 900

radiance, it may be worthwhile to read the paper by Villemain and Hery (2013) on Monte Carlo sampling of 3D emissive volumes.

- ③ 15.7 Compare rendering subsurface scattering with a BSSRDF to brute force integration of the same underlying medium properties with the `VolPathIntegrator`. (Recall that in high-albedo media, paths of hundreds or thousands of bounces may be necessary to compute accurate results.) Compare scenes with a variety of scattering properties, including both low and high albedos. Render images that demonstrate cases where the BSSRDF approximation introduces noticeable error but Monte Carlo computes a correct result. How much slower is the Monte Carlo approach for cases where the BSSRDF is accurate?
- ③ 15.8 Donner et al. (2009) performed extensive numerical simulation of subsurface scattering from media with a wide range of scattering properties and then computed coefficients to fit an analytical model to the resulting data. They have shown that rendering with this model is more efficient than full Monte Carlo integration, while handling well many cases where the approximations of many BSSRDF models are unacceptable. For example, their model accounts for directional variation in the scattered radiance and handles media with low and medium albedos well. Read their paper and download the data files of coefficients. Implement a new BSSRDF in `pbrt` that uses their model, and render images showing cases where it gives better results than the current BSSRDF implementation.



CHAPTER SIXTEEN

★ 16 LIGHT TRANSPORT III: BIDIRECTIONAL METHODS

The integrators in the previous two chapters have all been based on finding light-carrying paths starting from the camera and then only trying to connect with light sources at the last vertices of the paths. This chapter introduces algorithms based on sampling paths starting from both the camera and the lights and then connecting them at intermediate vertices. These algorithms can be much more efficient at finding light-carrying paths than approaches that only construct paths from the camera, especially in tricky lighting situations.

The foundations of bidirectional light transport are fascinating. On one hand, the physics of light scattering are generally reversible with respect to the direction of light transport, which causes the mathematical expressions of scattering paths starting from a light or from the camera to be very similar. On the other hand, there are subtle but important differences between these two approaches depending the path direction; Section 16.1 discusses these topics in detail. After the foundations have been set, the stochastic progressive photon mapping (SPPM) algorithm is introduced in Section 16.2. SPPM allows light-carrying particles to provide incident illumination at points close to where they intersect surfaces and not just exactly at their intersection points; this adjustment introduces bias but improves the rate of convergence in many challenging settings.

Next, bidirectional path tracing is introduced in Section 16.3. This unbiased approach can be much more efficient than regular path tracing by virtue of both its bidirectional nature as well as further variance reduction from applying multiple importance sampling to reweight path contributions. Finally, in Section 16.4, we show how Metropolis sampling (introduced in Section 13.4) can be used to further improve the efficiency of bidirectional path tracing by focusing computational effort on the most important light-carrying paths.

16.1 THE PATH-SPACE MEASUREMENT EQUATION

In light of the path integral form of the LTE from Equation (14.16), it's useful to go back and formally describe the quantity that is being estimated when we compute pixel values for an image. Not only does this let us see how to apply the LTE to a wider set of problems than just computing 2D images (e.g., to precomputing scattered radiance distributions at the vertices of a polygonal model), but this process also leads us to a key theoretical mechanism for understanding the bidirectional path tracing and photon mapping algorithms in this chapter. For simplicity, we'll use the basic path integral for surfaces rather than the generalized variant from Section 15.1.1, though the conclusions are applicable to both versions of the LTE.

The *measurement equation* describes the value of an abstract measurement that is found by integrating over some set of rays carrying radiance.¹ For example, when computing the value of a pixel j 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:

$$\begin{aligned} I_j &= \int_{A_{\text{film}}} \int_{S^2} W_e^{(j)}(\mathbf{p}_{\text{film}}, \omega) L_i(\mathbf{p}_{\text{film}}, \omega) |\cos \theta| d\omega dA(\mathbf{p}_{\text{film}}) \\ &= \int_{A_{\text{film}}} \int_A W_e^{(j)}(\mathbf{p}_0 \rightarrow \mathbf{p}_1) L(\mathbf{p}_1 \rightarrow \mathbf{p}_0) G(\mathbf{p}_0 \leftrightarrow \mathbf{p}_1) dA(\mathbf{p}_1) dA(\mathbf{p}_0), \end{aligned}$$

where I_j is the measurement for the j th pixel and \mathbf{p}_0 is a point on the film. In this setting, the $W_e^{(j)}(\mathbf{p}_0 \rightarrow \mathbf{p}_1)$ 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 \mathbf{p}_0 , $\omega_{\text{camera}}(\mathbf{p}_1)$:

$$W_e^{(j)}(\mathbf{p}_0 \rightarrow \mathbf{p}_1) = f_j(\mathbf{p}_0) \delta(t(\mathbf{p}_0, \omega_{\text{camera}}(\mathbf{p}_1)) - \mathbf{p}_1).$$

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

$$\begin{aligned} I_j &= \int_{A_{\text{film}}} \int_A W_e^{(j)}(\mathbf{p}_0 \rightarrow \mathbf{p}_1) L(\mathbf{p}_1 \rightarrow \mathbf{p}_0) G(\mathbf{p}_0 \leftrightarrow \mathbf{p}_1) dA(\mathbf{p}_1) dA(\mathbf{p}_0) \\ &= \sum_i \int_A \int_A W_e^{(j)}(\mathbf{p}_0 \rightarrow \mathbf{p}_1) P(\bar{\mathbf{p}}_i) G(\mathbf{p}_0 \leftrightarrow \mathbf{p}_1) dA(\mathbf{p}_1) dA(\mathbf{p}_0) \\ &= \sum_i \underbrace{\int_A \cdots \int_A}_{i+1 \text{ times}} W_e^{(j)}(\mathbf{p}_0 \rightarrow \mathbf{p}_1) T(\bar{\mathbf{p}}_i) L_e(\mathbf{p}_{i+1} \rightarrow \mathbf{p}_i) G(\mathbf{p}_0 \leftrightarrow \mathbf{p}_1) dA(\mathbf{p}_{i+1}) \cdots dA(\mathbf{p}_0), \end{aligned} \quad [16.1]$$

where $T(\bar{\mathbf{p}}_i)$ is the path throughput function introduced in Equation (14.18).

¹ The camera measurement equation described in Section 6.4.7 is a specific case of the measurement equation.

Note the nice symmetric way in which the emitted radiance L_e (quantifying the light source emission profile) and the weighting function $W_e^{(j)}$ (quantifying the camera's sensitivity profile for a pixel j) appear in the above equation: neither term is treated specially, and from this we can infer that the concepts of emission and measurement are mathematically interchangeable.

The implications of this symmetry are important: it says that we can think of the rendering process in two different ways. The first interpretation is that 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 the sensor as emitting an imaginary quantity that creates a measurement when it reaches a light source. This idea isn't just a theoretical construct: it can be applied in practice. A good example is the Dual Photography work of Sen et al. (2005) that showed that it was possible to take photographs from the viewpoint of a video projector by processing input photographs taken with a separate camera—this could be interpreted as turning the projector into a camera, while using the original camera as the “light source” to illuminate the scene.

By simply swapping the role of cameras and light sources in this way, we can create a method known as *particle tracing*, which traces rays from the light sources to recursively estimate the incident importance arriving on surfaces. This is not a particularly useful rendering technique on its own, but it constitutes an essential ingredient of other methods such as bidirectional path tracing and photon mapping.

The value described by the W_e term is known as the *importance* for the ray between p_0 and p_1 in the scene.² 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 measurements 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 also the integral that must be estimated to compute them.

16.1.1 SAMPLING CAMERAS

Bidirectional light transport algorithms require the ability to evaluate the value of importance function for arbitrary points in the scene; this is useful, for example, for computing the importance of a point along a path that started at a light source. The `Camera::We()` method takes a ray with origin p and direction ω and evaluates the importance emitted from the point on the camera p in a direction ω . If provided, the `pRaster2` parameter is used to return the raster position associated with the ray on the film; conceptually this can be understood as the discrete index j such that $W_e^{(j)}(p, \omega)$ attains its maximum value; in practice, the function returns fractional values to specify the raster positions more accurately.

(Camera Interface) +≡

356

```
virtual Spectrum We(const Ray &ray, Point2f *pRaster2 = nullptr) const;
```

Point2f 68

Ray 73

Spectrum 315

2 Note the overloaded terminology: the importance emanated by a camera is not related to the concept of importance sampling a statistical distribution; it will generally be clear which of the two is meant. However, we will later combine these concepts to implement code that importance samples a camera's importance function.

The default implementation of this method generates an error message; it's currently only implemented for the perspective camera model in pbrt. Implementing it for other Camera models is saved for Exercise 16.1 at the end of the chapter.

(PerspectiveCamera Method Definitions) +≡

```
Spectrum PerspectiveCamera::We(const Ray &ray, Point2f *pRaster2) const {
    (Interpolate camera matrix and check if ω is forward-facing 950)
    (Map ray (p, ω) onto the raster grid 950)
    (Return raster position if requested 950)
    (Return zero importance for out of bounds points 951)
    (Compute lens area of perspective camera 953)
    (Return importance for point on image plane 953)
}
```

Given the camera-to-world transformation for the provided time, the method checks that the direction ω points in the same hemisphere as the camera is facing by transforming the camera-space viewing direction $(0, 0, 1)$ to world space and computing the cosine of the angles between them. If these directions are more than 90 degrees apart, then the camera would never return a ray in this direction from its `GenerateRay()` method, and so an importance value of 0 can be returned immediately.

(Interpolate camera matrix and check if ω is forward-facing) ≡

950

```
Transform c2w;
CameraToWorld.Interpolate(ray.time, &c2w);
Float cosTheta = Dot(ray.d, c2w(Vector3f(0, 0, 1)));
if (cosTheta <= 0)
    return 0;
```

A slightly more involved test next checks if the ray corresponds to one starting from the film area. If its origin is outside of the film's extent, then the point p is outside of the camera's viewing volume, and again a zero importance value can be returned.

For a camera with a finite aperture, we have a point on the lens and its direction (Figure 16.1). We don't yet know the point on the film that this ray corresponds to, but we do know that all rays leaving that point are in focus at the plane $z = \text{focalDistance}$. Therefore, if we compute the ray's intersection with the plane of focus, then transforming that point with the perspective projection matrix gives us the corresponding point on the film. For pinhole apertures, we compute the intersection with a plane arbitrarily set at $z = 1$ to get a point along the ray leaving the camera before performing the projection.

(Map ray (p, ω) onto the raster grid) ≡

950, 953

```
Point3f pFocus = ray((lensRadius > 0 ? focalDistance : 1) / cosTheta);
Point3f pRaster = Inverse(RasterToCamera)(Inverse(c2w)(pFocus));
```

(Return raster position if requested) ≡

950

```
if (pRaster2 *pRaster2 = Point2f(pRaster.x, pRaster.y);
```

Given the raster-space point, it's easy to check if it's inside the image extent.

```
AnimatedTransform::  
    Interpolate()  
    106  
  
Camera 356  
Camera::CameraToWorld 356  
Dot() 63  
Float 1062  
Interaction::time 115  
Inverse() 1081  
Point2f 68  
Point3f 68  
ProjectiveCamera::  
    focalDistance  
    374  
ProjectiveCamera::lensRadius  
    374  
ProjectiveCamera::  
    RasterToCamera  
    360  
Ray 73  
Spectrum 315  
Transform 83  
Vector3f 60
```

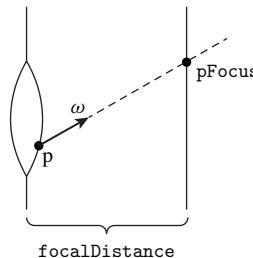


Figure 16.1: Computing the Focus Point p_{Focus} for a Ray Leaving the Lens. In order to compute the value of the importance function W_e for this ray, we need to find the point it corresponds to on the film plane. To do so, we first compute p_{Focus} , the point where the ray intersects the plane of focus. This point can in turn be projected by the camera's perspective projection matrix to find the corresponding raster-space point on the film.

(Return zero importance for out of bounds points) \equiv

```
Bounds2i sampleBounds = film->GetSampleBounds();
if (pRaster.x < sampleBounds.pMin.x || pRaster.x >= sampleBounds.pMax.x ||
    pRaster.y < sampleBounds.pMin.y || pRaster.y >= sampleBounds.pMax.y)
    return 0;
```

The perspective camera in pbrt is a ideal sensor in the sense that it generates samples with a uniform distribution over the film area. We will now use this fact to derive the corresponding directional sampling distribution. We'll start by defining a camera space image rectangle that all camera rays pass through and (arbitrarily) choose the one on the plane $z = 1$. The following fragment that is part of the `PerspectiveCamera` constructor uses the `RasterToCamera` transformation and divides by the z coordinate to compute the rectangle's corner points, which in turn gives the rectangle's area A .

(Compute image plane bounds at $z = 1$ for PerspectiveCamera) \equiv

```
Bounds2i 76
Camera::film 356
Film::fullResolution 485
Film::GetSampleBounds() 487
Float 1062
PerspectiveCamera 365
PerspectiveCamera:::
    GenerateRay()
    367
Point2i 68
Point3f 68
ProjectiveCamera:::
    RasterToCamera
    360
```

(PerspectiveCamera Private Data) \equiv

Float A;

The importance function doesn't have to obey any normalization constraints (just like emitted radiance from an area light). However, we'll define the `PerspectiveCamera`'s importance function as a normalized PDF on ray space, both for convenience in the following and also to be consistent with the weight values of 1 returned from `PerspectiveCamera::GenerateRay()`.

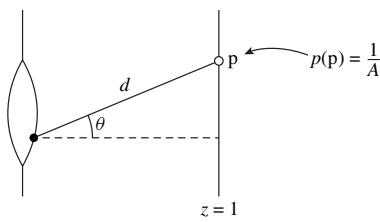


Figure 16.2: Deriving the Normalized Importance Function for the PerspectiveCamera. Given a point on the visible region of the image plane at $z = 1$ with PDF $p(p) = 1/A$, we can compute the directional PDF at a point on the lens (filled circle) by applying Equation (5.6) to account for the distance to the point on the lens d and θ , the angle between the vector from p to the point on the lens, and the surface normal of the image plane.

The importance function of `PerspectiveCamera` varies smoothly over its support ($W_e(\omega) > 0$); this variation is defined so that it cancels the vignetting that a real pinhole camera would have and ensures that pixels record values in units of radiance (this is another reason why `PerspectiveCamera::GenerateRay()` returns weight values of 1).

The camera uniformly generates samples over the image plane area A ; thus, the area-measured PDF for points on the image plane is $p(p) = 1/A$. Consider now the directional PDF at a point on the lens (or the camera's pinhole) corresponding to a differential area on the image plane (Figure 16.2); applying Equation (5.6) to transform to a directional density gives

$$p(\omega) = \begin{cases} \frac{d^2}{A \cos \theta}, & \text{if } \omega \text{ is within the frustum} \\ 0, & \text{otherwise,} \end{cases}$$

where θ is the angle that ω makes with the image rectangle normal and d is the distance between the point on the lens and the intersection of the ray with the $z = 1$ plane. The distance to the point on the image plane is

$$d = \left\| \frac{\omega}{\cos \theta} \right\| = \frac{1}{\cos \theta},$$

as $\cos \theta$ is the z coordinate of ω in the local camera coordinate system and $\|\omega\| = 1$. Hence,

$$p(\omega) = \begin{cases} \frac{1}{A \cos^3 \theta}, & \text{if } \omega \text{ is within the frustum} \\ 0, & \text{otherwise.} \end{cases} \quad [16.2]$$

By construction, the above density function $p(\omega)$ is normalized when integrated over directions—however, we initially set out to create a normalized importance function $W_e(p, \omega)$ that is defined on the space of camera rays $A_{\text{lens}} \times \mathbb{S}^2$, where A_{lens} is the surface region associated with the perspective camera's lens element. This function must satisfy the ray-space normalization criterion

$$\int_{A_{\text{lens}}} \int_{\mathbb{S}^2} W_e(p, \omega) |\cos \theta| d\omega dA(p) = 1. \quad [16.3]$$

We can't directly set $W_e(p, \omega)$ equal to $p(\omega)$ due to the extra integration over areas and the additional cosine factor in the above integral.

`PerspectiveCamera` 365

`PerspectiveCamera::GenerateRay()`
367

Note that the lens area A_{lens} of a perspective camera is equal to πr^2 , where r is the lens radius. For point camera, the lens area is set to 1 and interpreted as a Dirac delta function.

(Compute lens area of perspective camera) \equiv 950, 953, 955
`Float lensArea = lensRadius != 0 ? (Pi * lensRadius * lensRadius) : 1;`

We then define $W_e(p, \omega)$ on ray space as

$$W_e(p, \omega) = \frac{p(\omega)}{\pi r^2 \cos \theta} = \begin{cases} \frac{1}{A \pi r^2 \cos^4 \theta}, & \text{if } \omega \text{ is within the frustum} \\ 0, & \text{otherwise,} \end{cases} \quad (16.4)$$

which divides p by the lens area and a term to cancel out the cosine factor from Equation (16.3). At this point, the implementation has already ensured that ω is within the frustum, hence the only thing left to do is to return the importance value according to the first case of Equation (16.4).

(Return importance for point on image plane) \equiv 950
`Float cos2Theta = cosTheta * cosTheta;
return Spectrum(1 / (A * lensArea * cos2Theta * cos2Theta));`

In the light of a definition for the perspective camera's importance function, we can now reinterpret the ray generation function `Camera::GenerateRay()` as an importance sampling technique for `W_e()`. As such, it's appropriate to define a `Camera` method that separately returns the spatial and directional PDFs for sampling a particular ray leaving the camera, analogous to the `Light::Pdf_Le()` method for light sources that will be introduced in Section 16.1.2. As before, this method is currently only implemented for the `PerspectiveCamera` and the default implementation generates an error message.

(Camera Interface) \equiv 356
`virtual void Pdf_We(const Ray &ray, Float *pdfPos, Float *pdfDir) const;`

The directional density of the ideal sampling strategy implemented in the method `PerspectiveCamera::GenerateRay()` was already discussed and is equal to $p(\omega)$ defined in Equation (16.2). The spatial density is the reciprocal of the lens area. Due to this overlap, the first four fragments of `PerspectiveCamera::Pdf_We()` are therefore either identical or almost identical to the similarly named fragments from `PerspectiveCamera::We()` with the exception that they return zero probabilities via `*pdfPos` and `*pdfDir` upon failure.

(PerspectiveCamera Method Definitions) \equiv
`void PerspectiveCamera::Pdf_We(const Ray &ray, Float *pdfPos,
Float *pdfDir) const {`
(Interpolate camera matrix and fail if ω is not forward-facing)
(Map ray (p, ω) onto the raster grid 950)
(Return zero probability for out of bounds points)
(Compute lens area of perspective camera 953)
`*pdfPos = 1 / lensArea;`
`*pdfDir = 1 / (A * cosTheta * cosTheta * cosTheta);`
`}`

Camera 356
`Camera::GenerateRay()` 357
Float 1062
`Light::Pdf_Le()` 955
PerspectiveCamera 365
`PerspectiveCamera::A` 951
`PerspectiveCamera::GenerateRay()`
367
`PerspectiveCamera::Pdf_We()` 953
`PerspectiveCamera::We()` 950
Pi 1063
`ProjectiveCamera::lensRadius`
374
Ray 73
Spectrum 315

One last additional Camera method completes the symmetry between light sources and cameras: it samples a point on the camera lens and computes an incident direction along which importance is arriving at the given reference position in the scene; it is thus the camera equivalent of Light::Sample_Li().

Like Sample_Li(), the PDF value this method returns is defined with respect to solid angle at the reference point.

(Camera Interface) +≡

```
virtual Spectrum Sample_Wi(const Interaction &ref, const Point2f &u,
                           Vector3f *wi, Float *pdf, Point2f *pRaster,
                           VisibilityTester *vis) const;
```

356

The PerspectiveCamera implementation of this method samples a point on the lens to compute the incident importance at the reference point.

(PerspectiveCamera Method Definitions) +≡

```
Spectrum PerspectiveCamera::Sample_Wi(const Interaction &ref,
                                       const Point2f &u, Vector3f *wi, Float *pdf, Point2f *pRaster,
                                       VisibilityTester *vis) const {
    (Uniformly sample a lens interaction lensIntr 954)
    (Populate arguments and compute the importance value 954)
}
```

We can compute an Interaction for a point on the lens by using u to sample the lens and then transforming this point to world space. For pinhole cameras, lensRadius is 0 and pLens always ends up being at the origin.

(Uniformly sample a lens interaction lensIntr) ≡

```
Point2f pLens = lensRadius * ConcentricSampleDisk(u);
Point3f pLensWorld =
    CameraToWorld(ref.time, Point3f(pLens.x, pLens.y, 0));
Interaction lensIntr(pLensWorld, ref.time, medium);
lensIntr.n = Normal3f(CameraToWorld(ref.time, Vector3f(0, 0, 1)));
```

954

Camera 356
 Camera::CameraToWorld 356
 Camera::medium 356
 ConcentricSampleDisk() 778
 Float 1062
 Interaction 115
 Interaction::n 116
 Interaction::p 115
 Interaction::SpawnRay() 232
 Interaction::time 115
 Light::Sample_Li() 716
 Normal3f 71
 PerspectiveCamera 365
 PerspectiveCamera::We() 950
 Point2f 68
 Point3f 68
 ProjectiveCamera::lensRadius 374
 Spectrum 315
 Vector3f::Length() 65
 Vector3f 60
 VisibilityTester 717

Given the point on the lens, most of the output parameters of Sample_We() can be initialized in a straightforward manner.

(Populate arguments and compute the importance value) ≡

```
*vis = VisibilityTester(ref, lensIntr);
*wi = lensIntr.p - ref.p;
Float dist = wi->Length();
*wi /= dist;
(Compute PDF for importance arriving at ref 955)
return We(lensIntr.SpawnRay(-*wi), pRaster);
```

954

The PDF of the sample is the probability of sampling a point on the lens ($1 / \text{lensArea}$), converted into a probability per unit solid angle at the reference point. For pinhole cameras, there is an implied delta distribution in both the PDF and the importance function value that cancel out later. (This is following the same convention as was used for BSDFs and light sources in Sections 14.1.3 and 14.2.1).

```
<Compute PDF for importance arriving at ref> ≡ 954
<Compute lens area of perspective camera 953>
*pdf = (dist * dist) / (AbsDot(lensIntr.n, *wi) * lensArea);
```

16.1.2 SAMPLING LIGHT RAYS

For bidirectional light transport algorithms, it's also necessary to add a light sampling method, `Sample_Le()`, that samples a ray from a distribution of rays *leaving* the light, returning the ray in `*ray` and the surface normal at the point on the light source in `*nLight` (effectively, the analog of `Camera::GenerateRay()`). A total of four sample values are passed to this method in the `u1` and `u2` parameters so that two are available to sample the ray's origin and two are available for its direction. Not all light implementations need all of these values—for example, the origin of all rays leaving a point light is the same.

This method returns two PDF values: the ray origin's probability density with respect to surface area on the light and its direction's probability density with respect to solid angle. The joint probability of sampling the ray is the product of these two probabilities.

```
<Light Interface> +≡ 714
virtual Spectrum Sample_Le(const Point2f &u1, const Point2f &u2,
                           Float time, Ray *ray, Normal3f *nLight,
                           Float *pdfPos, Float *pdfDir) const = 0;
```

So that multiple importance sampling can be applied, there is also a method to return the position and direction PDFs for a given ray.

```
<Light Interface> +≡ 714
virtual void Pdf_Le(const Ray &ray, const Normal3f &nLight,
                     Float *pdfPos, Float *pdfDir) const = 0;
```

Point Lights

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

```
<PointLight Method Definitions> +≡
Spectrum PointLight::Sample_Le(const Point2f &u1, const Point2f &u2,
                               Float time, Ray *ray, Normal3f *nLight, Float *pdfPos,
                               Float *pdfDir) const {
    *ray = Ray(pLight, UniformSampleSphere(u1), Infinity, time,
               mediumInterface.inside);
    *nLight = (Normal3f)ray->d;
    *pdfPos = 1;
    *pdfDir = UniformSpherePdf();
    return I;
}
```

AbsDot() 64
`Camera::GenerateRay()` 357
`Float` 1062
`Infinity` 210
`Interaction::n` 116
`Light::mediumInterface` 715
`MediumInterface::inside` 684
`Normal3f` 71
`Point2f` 68
`PointLight::I` 720
`PointLight::pLight` 720
`Ray` 73
`Spectrum` 315
`UniformSampleSphere()` 776
`UniformSpherePdf()` 776

```
(PointLight Method Definitions) +≡
void PointLight::Pdf_Le(const Ray &, const Normal3f &, Float *pdfPos,
                      Float *pdfDir) const {
    *pdfPos = 0;
    *pdfDir = UniformSpherePdf();
}
```

Spotlights

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

The PDF $p(\theta, \phi)$ for the spotlight's illumination distribution is separable with $p(\phi) = 1/(2\pi)$. We thus just need to find a sampling distribution for θ . The `UniformSampleCone()` function from Section 13.6.4 provides this functionality.

(SpotLight Method Definitions) +≡

```
Spectrum SpotLight::Sample_Le(const Point2f &u1, const Point2f &u2,
                           Float time, Ray *ray, Normal3f *nLight, Float *pdfPos,
                           Float *pdfDir) const {
    Vector3f w = UniformSampleCone(u1, cosTotalWidth);
    *ray = Ray(pLight, LightToWorld(w), Infinity, time,
               mediumInterface.inside);
    *nlight = (Normal3f)ray->d;
    *pdfPos = 1;
    *pdfDir = UniformConePdf(cosTotalWidth);
    return I * Falloff(ray->d);
}
```

CosTheta() 510
 Float 1062
 GonioPhotometricLight 728
 Infinity 210
 Light::LightToWorld 715
 Light::mediumInterface 715
 Light::WorldToLight 715
 MediumInterface::inside 684
 Normal3f 71
 Point2f 68
 PointLight 719
 ProjectionLight 724
 ProjectionLight::cosTotalWidth 727
 Ray 73
 Spectrum 315
 SpotLight 721
 SpotLight::cosTotalWidth 723
 SpotLight::Falloff() 724
 SpotLight::I 723
 UniformConePdf() 781
 UniformSampleCone() 781
 UniformSpherePdf() 776
 Vector3f 60

The `SpotLight`'s `Pdf_Le()` method for sampled rays must check to see if the direction is inside the cone of illuminated directions before returning the cone sampling PDF.

(SpotLight Method Definitions) +≡

```
void SpotLight::Pdf_Le(const Ray &ray, const Normal3f &, Float *pdfPos,
                      Float *pdfDir) const {
    *pdfPos = 0;
    *pdfDir = (CosTheta(WorldToLight(ray.d)) >= cosTotalWidth) ?
        UniformConePdf(cosTotalWidth) : 0;
}
```

The sampling routines for `ProjectionLights` and `GonioPhotometricLights` are essentially the same as the ones for `SpotLights` and `PointLights`, respectively. For sampling outgoing rays, `ProjectionLights` sample uniformly from the cone that encompasses their projected image map (hence the need to compute `ProjectionLight::cosTotalWidth` in the constructor), and those for `GonioPhotometricLights` sample uniformly over the unit

sphere. Exercise 16.2 at the end of this chapter discusses improvements to these sampling methods that better account for the directional variation of these lights.

Area Lights

The method for sampling a ray leaving an area light is also easily implemented in terms of the shape sampling methods from Section 14.2.2.

```
(DiffuseAreaLight Method Definitions) +≡
Spectrum DiffuseAreaLight::Sample_Le(const Point2f &u1, const Point2f &u2,
    Float time, Ray *ray, Normal3f *nLight, Float *pdfPos,
    Float *pdfDir) const {
    (Sample a point on the area light's Shape, pShape 957)
    (Sample a cosine-weighted outgoing direction w for area light 957)
    *ray = pShape.SpawnRay(w);
    return L(pShape, w);
}
```

The surface area-based variant of `Shape::Sample()` is used to find the ray origin, sampled from some density over the surface.

```
(Sample a point on the area light's Shape, pShape) ≡
Interaction pShape = shape->Sample(u1);
pShape.mediumInterface = mediumInterface;
*pdfPos = shape->Pdf(pShape);
*nLight = pShape.n;
```

957

The ray's direction is sampled from a cosine-weighted distribution about the surface normal at the sampled point. Incorporating this cosine weighting means that rays leaving the light carry uniform differential power, which is preferable for bidirectional light transport algorithms. Because the direction returned by `CosineSampleHemisphere()` is in the canonical coordinate system, it must be transformed to the coordinate system about the surface normal at the sampled point here.

```
(Sample a cosine-weighted outgoing direction w for area light) ≡
Vector3f w = CosineSampleHemisphere(u2);
*pdfDir = CosineHemispherePdf(w.z);
Vector3f v1, v2, n(pShape.n);
CoordinateSystem(n, &v1, &v2);
w = w.x * v1 + w.y * v2 + w.z * n;
```

957

Distant Lights

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

CoordinateSystem() 67
CosineHemispherePdf() 780
CosineSampleHemisphere() 780
DiffuseAreaLight::L() 736
DiffuseAreaLight::shape 736
DistantLight 731
Float 1062
Interaction 115
Interaction::mediumInterface 116
Interaction::n 116
Interaction::SpawnRay() 232
Light::mediumInterface 715
Normal3f 71
Point2f 68
Ray 73
Shape::Pdf() 837
Shape::Sample() 837
Spectrum 315
Vector3f 60

The desired property is that rays intersect points in the scene that are illuminated by the distant light with uniform probability. One way to do this is to construct a disk that has the same radius as the scene's bounding sphere and has a normal that is oriented with the light's direction and then choose a random point on this disk, using the

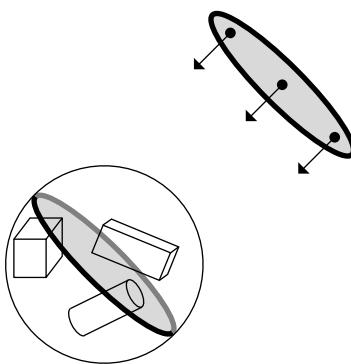


Figure 16.3: To sample an outgoing ray direction for a distant light source, the `DistantLight::Sample_Le()` method finds the disk oriented in the light's direction that is large enough so that the entire scene can be intersected by rays leaving the disk in the light's direction. Ray origins are sampled uniformly by area on this disk, and ray directions are given directly by the light's direction.

`ConcentricSampleDisk()` function (Figure 16.3). Once this point has been chosen, if the point is displaced along the light's direction by the scene's bounding sphere radius and used as the origin of the light ray, the ray origin will be outside the bounding sphere of the scene.

This is a valid sampling approach, since by construction it has nonzero probability of sampling all incident rays into the sphere due to the directional light. The area component of the sampling density is uniform and therefore equal to the reciprocal of the area of the disk that was sampled. The directional density is given by a delta distribution based on the light's direction.

(DistantLight Method Definitions) +≡

```
Spectrum DistantLight::Sample_Le(const Point2f &u1, const Point2f &u2,
    Float time, Ray *ray, Normal3f *nLight, Float *pdfPos,
    Float *pdfDir) const {
    (Choose point on disk oriented toward infinite light direction 959)
    (Set ray origin and direction for infinite light ray 959)
    *nLight = (Normal3f)ray->d;
    *pdfPos = 1 / (Pi * worldRadius * worldRadius);
    *pdfDir = 1;
    return L;
}
```

`ConcentricSampleDisk()` 778

`DistantLight::worldRadius` 732

`Float` 1062

`Normal3f` 71

`Pi` 1063

`Point2f` 68

`Ray` 73

`Spectrum` 315

Choosing the point on the oriented disk is a simple application of vector algebra. We construct a coordinate system with two vectors perpendicular to the disk's normal (the light's direction); see Figure 16.4. Given a random point on the canonical unit disk, computing the offsets from the disk's center with respect to its coordinate vectors gives the corresponding point.

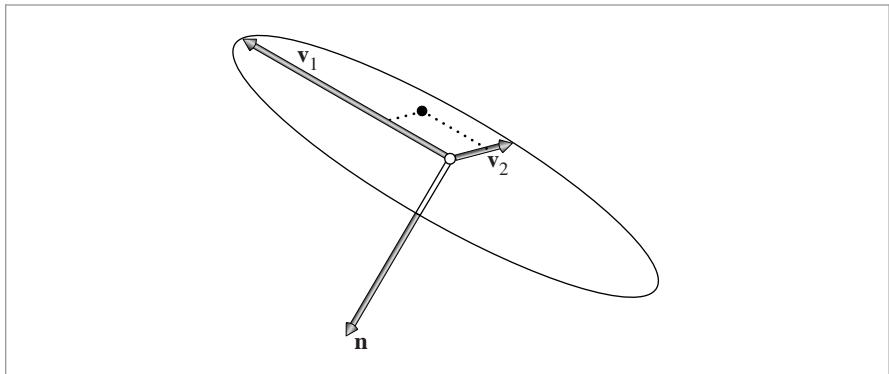


Figure 16.4: Given sample points (d_1, d_2) on the canonical unit disk, points on an arbitrarily oriented and sized disk with normal \mathbf{n} can be found by computing an arbitrary coordinate system $(\mathbf{v}_1, \mathbf{v}_2, \mathbf{n})$ and then computing points on the disk with the offset $d_1\mathbf{v}_1 + d_2\mathbf{v}_2$ from the disk's center.

(Choose point on disk oriented toward infinite light direction) \equiv

958

```
Vector3f v1, v2;
CoordinateSystem(wLight, &v1, &v2);
Point2f cd = ConcentricSampleDisk(u1);
Point3f pDisk = worldCenter + worldRadius * (cd.x * v1 + cd.y * v2);
```

Finally, the point is offset along the light direction and the ray can be initialized. Recall from Section 12.4 that *DistantLights* can't be embedded in any medium other than a vacuum. Therefore, no medium needs to be specified for the ray.

(Set ray origin and direction for infinite light ray) \equiv

958

```
*ray = Ray(pDisk + worldRadius * wLight, -wLight, Infinity, time);
```

Infinite Area Lights

Generating a random ray leaving an infinite light source can be done by sampling a direction with the same approach as the earlier *InfiniteAreaLight::Sample_Li()* method. The sampled ray's origin is then set using the same approach as was used for *DistantLights*, where a disk that covers the scene's bounding sphere is oriented along the ray's direction (recall Figure 16.3). We therefore won't include the *(Compute direction for infinite light sample ray)* or *(Compute origin for infinite light sample ray)* fragments here.

(InfiniteAreaLight Method Definitions) \equiv

```
Spectrum InfiniteAreaLight::Sample_Le(const Point2f &u1,
    const Point2f &u2, Float time, Ray *ray, Normal3f *nLight,
    Float *pdfPos, Float *pdfDir) const {
    (Compute direction for infinite light sample ray)
    (Compute origin for infinite light sample ray)
    (Compute InfiniteAreaLight ray PDFs 960)
    return Spectrum(Lmap->Lookup(uv), SpectrumType::Illuminant);
}
```

ConcentricSampleDisk() 778
 CoordinateSystem() 67
 DistantLight 731
 DistantLight::wLight 731
 DistantLight::worldCenter 732
 DistantLight::worldRadius 732
 Float 1062
 InfiniteAreaLight::Lmap 740
 InfiniteAreaLight:
 Sample_Li()
 849
 Infinity 210
 MIPMap::Lookup() 635
 Normal3f 71
 Point2f 68
 Point3f 68
 Ray 73
 Spectrum 315
 SpectrumType::Illuminant 330
 Vector3f 60

The PDFs for these rays are the PDF for sampling the direction (as derived in Section 14.2.4) and the PDF for sampling a point on the disk.

```
(Compute InfiniteAreaLight ray PDFs) ≡ 959
  *pdfDir = sinTheta == 0 ? 0 : mapPdf / (2 * Pi * Pi * sinTheta);
  *pdfPos = 1 / (Pi * worldRadius * worldRadius);
```

The `Pdf_Le()` method applies the same formulas, so we won't include its implementation here.

16.1.3 NON-SYMMETRIC SCATTERING

Certain aspects in the input scene specification of materials and geometry can lead to non-symmetric behavior in light transport simulations, where incident radiance and importance are scattered in different ways at a point. If these differences aren't accounted for, rendering algorithms based on radiance and importance transport will produce different and inconsistent results when rendering the same input scene. Bidirectional techniques that combine radiance and importance transport are particularly affected, since their design is fundamentally based on the principle of symmetry.

In this section, we will briefly enumerate cases that give rise to non-symmetry and explain how they can be addressed to arrive at a consistent set of bidirectional estimators.

Recall the path throughput term $T(\bar{p}_i)$ from Equation (16.1), which was defined as

$$T(\bar{p}_n) = \prod_{i=1}^{n-1} f(p_{i+1} \rightarrow p_i \rightarrow p_{i-1}) G(p_{i+1} \leftrightarrow p_i).$$

The vertices are ordered such that p_i denotes the i -th scattering event as seen from the camera.

Sampling techniques based on finding importance-carrying paths trace rays starting at the light sources to estimate the incident importance at the light, which means that the vertices will be generated in reverse compared to the above ordering. As such, the incident and outgoing direction arguments of the BSDFs will be (incorrectly) reversed unless special precautions are taken. We thus define the *adjoint BSDF* f^* at vertex p_i , whose only role is to evaluate the original BSDF with swapped arguments:

$$f^*(p, \omega_o, \omega_i) = f(p, \omega_i, \omega_o).$$

All sampling steps based on importance transport will then use the adjoint form of the BSDF rather than its original version. Most BSDFs in pbrt are symmetric so that there is no actual difference between f and f^* . However, certain cases related to shading normals and light refracting into media with a different index of refraction require additional attention.

The `TransportMode` enumeration is used to inform such non-symmetric BSDFs about the transported quantity so that they can correctly switch between the adjoint and non-adjoint forms.

```
(TransportMode Declarations) ≡
  enum class TransportMode { Radiance, Importance };
```

```
InfiniteAreaLight::
worldRadius
740
Pi 1063
TransportMode 960
```

Non-symmetry Due to Refraction

When light refracts into a material with a higher index of refraction than the incident medium's index of refraction, the energy is compressed into a smaller set of angles. This is easy to see yourself, for instance, by looking at the sky from underwater in a quiet outdoor swimming pool. Because no light can be refracted below the critical angle ($\sim 48.6^\circ$ for water), the incident hemisphere of light is squeezed into a considerably smaller subset of the hemisphere, which covers the remaining set of angles. Radiance along rays that do refract must thus increase so that energy is preserved when light passes through the interface. More precisely, the incident (L_i) and transmitted (L_t) radiance are related by

$$L_i = \frac{\eta_i^2}{\eta_t^2} L_t, \quad (16.5)$$

where η_i and η_t are the refractive indices on the incident and transmitted sides, respectively. The symmetry relationship satisfied by a BTDF is

$$\eta_t^2 f(p, \omega_o, \omega_i) = \eta_i^2 f(p, \omega_i, \omega_o), \quad (16.6)$$

and we can obtain the adjoint BTDF

$$f^*(p, \omega_o, \omega_i) = f(p, \omega_i, \omega_o) = \frac{\eta_t^2}{\eta_i^2} f(p, \omega_o, \omega_i),$$

which effectively cancels out the scale factor in Equation (16.5). With these equations, we can now define the last missing piece in the implementation of `SpecularTransmission::Sample_f()`. Whenever radiance is transported over a refractive boundary, we apply the scale factor from Equation (16.5). For importance transport, we use the adjoint BTDF, which lacks the scaling factor due to the combination of Equations (16.5) and (16.6).

(Account for non-symmetry with transmission to different medium) ≡

529, 817

```
if (mode == TransportMode::Radiance)
    ft *= (etaI * etaI) / (etaT * etat);
```

A similar adjustment is also needed for the `FourierBSDF::f()` method in the case of refraction. In this case, `FourierBSDFTable::eta` provides the relative index of refraction. Recall that this model uses a convention where the sign of $\mu_i = \cos \theta_i$ is flipped, hence the expression $\mu I * \mu 0 > 0$ can be used to check if light is being refracted rather than reflected.

BSDF 572
BSSRDF::eta 692
FourierBSDF::f() 556
FourierBSDF::mode 556
FourierBSDFTable::eta 554
SeparableBSSRDF 693
SeparableBSSRDF::mode 693
SpecularTransmission::mode 529
SpecularTransmission::Sample_f() 529
TransportMode::Radiance 960

(Update scale to account for adjoint light transport) ≡

558

```
if (mode == TransportMode::Radiance && muI * mu0 > 0) {
    float eta = muI > 0 ? 1 / bsdfTable.eta : bsdfTable.eta;
    scale *= eta * eta;
}
```

Finally, the transmissive term S_ω of the `SeparableBSSRDF` requires a similar correction when light leaves the medium after a second refraction (the first one being handled by the material's BSDF).

(Update BSSRDF transmission term to account for adjoint light transport) ≡

906

```
if (bssrdf->mode == TransportMode::Radiance)
    f *= bssrdf->eta * bssrdf->eta;
```

Non-symmetry Due to Shading Normals

Shading normals are another cause of non-symmetric scattering. As previously discussed in Section 3.6.3, shading normals are mainly used to make polygonal surfaces appear smoother than their actual discretization. This entails replacing the “true” geometric normal \mathbf{n}_g with an interpolated shading normal \mathbf{n}_s whenever the BSDF or the cosine term in the light transport equation are evaluated. Bump or normal mapping can be interpreted as another kind of shading normal, where \mathbf{n}_s is obtained from a texture map.

This kind of modification to the normal of a surface interaction causes a corresponding change in the underlying reflectance model, producing an effective BSDF that is generally non-symmetric. Without additional precautions, this non-symmetry can lead to visible artifacts in renderings based on adjoint techniques, including discontinuities in shading effects resembling flat-shaded polygons that interpolated normals were originally meant to avoid.

Recall the light transport equation, (14.13), which relates the incident and outgoing radiance on surfaces:

$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + \int_{S^2} f(\mathbf{p}, \omega_o, \omega_i) L_i(\mathbf{p}, \omega_i) |\mathbf{n}_g \cdot \omega_i| d\omega_i.$$

Here, the cosine factor is expressed as an inner product involving ω_i and the true normal of the underlying geometry. Suppose now that we’d like to replace \mathbf{n}_g with the shading normal \mathbf{n}_s . Instead of modifying the scattering equation, another mathematically equivalent way of expressing this change entails switching to a new BSDF f_{shade} defined as

$$f_{\text{shade}}(\mathbf{p}, \omega_o, \omega_i) = \frac{|\mathbf{n}_s \cdot \omega_i|}{|\mathbf{n}_g \cdot \omega_i|} f(\mathbf{p}, \omega_o, \omega_i).$$

The first factor in the above expression makes this BSDF non-symmetric with respect to the arguments ω_i and ω_o . To avoid artifacts and inconsistencies in bidirectional rendering algorithms, the adjoint BSDF f_{shade}^* should be used in simulations whenever importance transport is used. It is given by

$$f_{\text{shade}}^*(\mathbf{p}, \omega_o, \omega_i) = \frac{|\mathbf{n}_s \cdot \omega_o|}{|\mathbf{n}_g \cdot \omega_o|} f^*(\mathbf{p}, \omega_o, \omega_i).$$

Rather than integrating this special case into all BxDF subclasses, we find it cleaner to detect this case in the integrator and apply a correction factor

$$C_{\text{shade}}(\mathbf{p}, \omega_o, \omega_i) = \begin{cases} \frac{|\mathbf{n}_s \cdot \omega_o| |\mathbf{n}_g \cdot \omega_i|}{|\mathbf{n}_g \cdot \omega_o| |\mathbf{n}_s \cdot \omega_i|} & \text{if importance is being transported} \\ 1 & \text{if radiance is being transported,} \end{cases}$$

which corrects the normal dependence of the non-adjoint version into that of the adjoint when importance transport is indicated by the mode parameter. This adjustment is implemented by the helper function `CorrectShadingNormal()` below.

BxDF 513

`CorrectShadingNormal()` 963

```

⟨BDPT Utility Functions⟩ ≡
    Float CorrectShadingNormal(const SurfaceInteraction &isect,
        const Vector3f &wo, const Vector3f &wi, TransportMode mode) {
        if (mode == TransportMode::Importance)
            return (AbsDot(wo, isect.shading.n) * AbsDot(wi, isect.n)) /
                (AbsDot(wo, isect.n) * AbsDot(wi, isect.shading.n));
        else
            return 1;
    }
}

```

16.2 STOCHASTIC PROGRESSIVE PHOTON MAPPING

Photon mapping is one of a family of particle-tracing algorithms, which are based on the idea of constructing paths starting from the lights and connecting vertices in these paths to the camera to deposit energy on the film. 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 close to but not quite at the point being shaded.

16.2.1 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

$$(p_j, \omega_j, \beta_j),$$

AbsDot() 64
 Float 1062
 Interaction::: 116
 SurfaceInteraction 116
 SurfaceInteraction:::
 shading::: 118
 TransportMode 960
 TransportMode::Importance 960
 Vector3f 60

where each sample records incident illumination from direction ω_j and has some throughput weight β_j associated with it (Figure 16.5). As the notation already indicates,

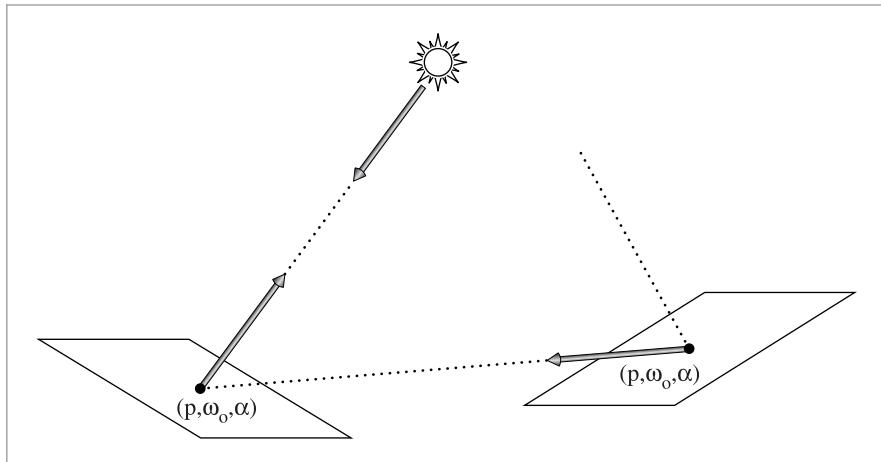


Figure 16.5: When a particle is traced following a path from a light source, an entry in its particle history is recorded at each surface it intersects. Each entry in the history is represented by position p , the direction of the ray it arrived along ω_o , and particle weight β .

this weight β_j will contain ratios of terms of the throughput function T and the associated sampling PDFs much like the β variable of the path tracer (Section 14.5.4). 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.

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:

$$E \left[\frac{1}{N} \sum_{j=1}^N \beta_j W_e(p_j, \omega_j) \right] = \int_A \int_{S^2} W_e(p, \omega) L_i(p, \omega) |\cos \theta| dA d\omega. \quad [16.7]$$

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

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

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

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

If the conditions on the distribution of particle weights and positions are true for arbitrary importance functions such that Equation (16.7) holds, then the flux estimate can be computed directly as a sum of the particle weights for the particles on the wall. 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 image artifacts.)

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

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

where the vertex densities $p(p_{i,j})$ are expressed as a probability per unit area and 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,

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

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

$$E \left[\frac{L(p_{i,1} \rightarrow p_{i,0})}{p(p_{i,0})} \right]$$

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 terminating the sum after j terms is $q_{i,j}$, then the j th term of the i th sample has contribution

$$\beta_{i,j} = \frac{L_e(p_{i,n_i} \rightarrow p_{i,n_i-1})}{p(p_{i,n_i})} \prod_{j=1}^{n_i-1} \frac{1}{1-q_{i,j}} \frac{f(p_{i,j+1} \rightarrow p_{i,j} \rightarrow p_{i,j-1}) G(p_{i,j+1} \leftrightarrow p_{i,j})}{p(p_{i,j})}.$$

(16.8)

Note that the path integral framework provides us with the freedom to generate a set of particles in all sorts of different ways—i.e., with different underlying vertex probability densities $p(p_{i,j})$. 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, as long as there is nonzero probability of generating a particle at any point where the numerator is nonzero and the particle weights $\beta_{i,j}$ are computed using the above definition.

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.

16.2.2 PHOTON MAPPING

The photon mapping algorithm is based on tracing particles into the scene and blurring their contribution 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.

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 :

$$\begin{aligned} & \int_{S^2} L_i(p, \omega_i) f(p, \omega_o, \omega_i) |\cos \theta_i| d\omega_i \\ &= \int_A \int_{S^2} \delta(p - p') L_i(p', \omega_i) f(p', \omega_o, \omega_i) |\cos \theta_i| d\omega_i dA(p'), \end{aligned}$$

and so, from Equation (16.7), the function that describes the measurement is

$$W_e(p', \omega) = \delta(p' - p) f(p, \omega_o, \omega). \quad [16.9]$$

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 (16.7) 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 based on the light source's position).

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 gives a reasonable approximation to illumination at the shading point, photon mapping interpolates illumination from nearby photons around the point being shaded; the delta function of position in Equation (16.9) is effectively converted to a filter function. Given Equation (16.7), the more photons there are around the point and the higher their weights, the more radiance is estimated to be incident at the point.

One factor that contributes to photon mapping's efficiency is this reuse of photons: having gone through the trouble to compute a light-carrying path, allowing it to potentially contribute illumination at multiple points amortizes the cost of generating it. While photon mapping derives some benefit from this efficiency improvement, there's a subtler but much more important benefit from using nearby photons at a point: some light-carrying paths are impossible to sample with unbiased algorithms based on incremental path con-

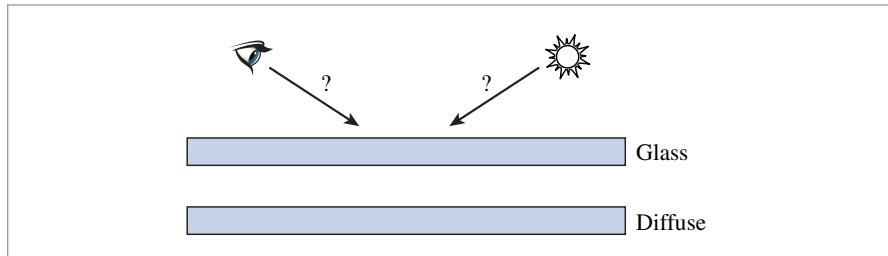


Figure 16.6: An Impossible-to-Sample Path for Path Tracing or Bidirectional Path Tracing. The scene includes a point light, a pinhole camera, and a diffuse surface behind a sheet of glass. Given a ray leaving the camera, the point at which it intersects the diffuse surface is determined based on refraction through the glass. At this point, only a single direction provides illumination from the point light, but there's no way to sample a direction leaving the surface that will intersect the light. The corresponding problem arises when starting the path from the light: there's no way to find a path back to the camera.

struction (including path tracing and bidirectional path tracing), but are handled well with photon mapping. These paths can arise in very common situations.

For example, consider the task of rendering an image of a photograph with a plate of glass in front of it. Assume a pinhole camera model and a point light illuminating the scene, and also assume for simplicity that the glass is only transmissive (Figure 16.6). If we start a path from the camera that passes through the glass, then the point that it intersects the photograph is completely determined by the effect of the refraction. At this point, none of the direct lighting strategies we have available has any chance of sampling an incident direction that will reach the light: because any sampled incident direction leaving the diffuse surface will be refracted on the way out through the glass, there's no chance that the refracted ray will hit the point light.³ Even with an area light source, some refracted rays may be lucky enough to hit the light, but in general, variance will be high since most will miss it.

With photon mapping, we can trace photons leaving the light, let them refract through the glass, and deposit illumination on the diffuse surface. With a sufficient number of photons, the surface will be densely covered, and the photons around a point give a good estimate of the incident illumination.

A statistical technique called *density estimation*⁴ 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

³ For a flat sheet of glass, one could implement a specialized sampling technique that accounted for this refraction, though doing so requires moving beyond the incremental path construction framework we've used so far. Such approaches become more challenging with more complex scene geometry.

⁴ Strictly speaking, density estimation can only be used to estimate the (normalized) density function of a set of unweighted samples. When working with weighted samples and general unnormalized functions, the term *kernel smoothing* is more commonly used. Although the latter case is the one relevant for photon mapping, we will continue refer to it as density estimation due to the heavy usage of this term in computer graphics.

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 1.

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

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* or *kernel bandwidth*). Kernel methods can be thought of as placing a series of bumps at observation points, where the sum of the bumps forms a PDF since they individually integrate to 1 and the sum is normalized. Figure 16.7 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

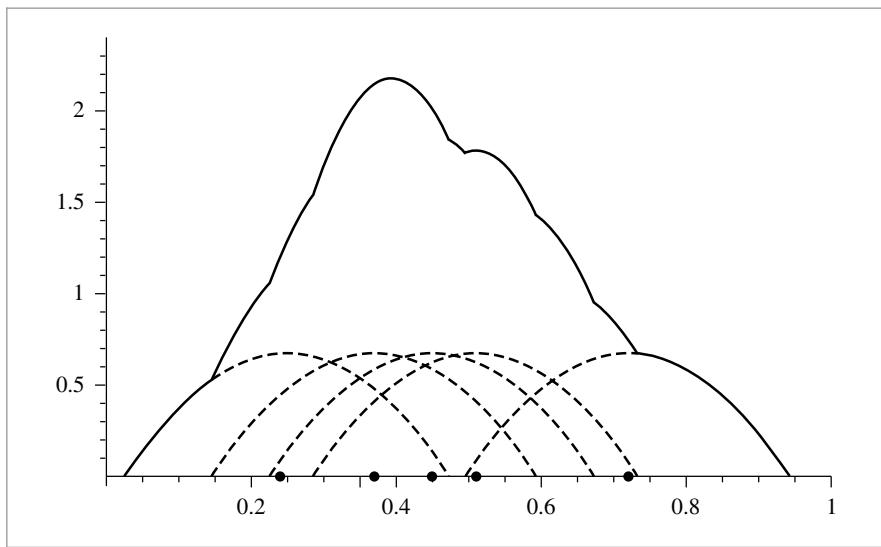


Figure 16.7: 1D example of density estimation, using the Epanechnikov kernel, $k(t) = 0.75(1 - .2t^2)/\sqrt{5}$, if $t < \sqrt{5}$, 0 otherwise, and a width of 0.1. The points marked with closed circles are the sample points, and an instance of the kernel (dashed lines) is placed over each one. The sum of the kernels gives a properly normalized PDF that attempts to model a distribution that the points could be distributed by.

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 N th 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*:

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

In d dimensions, this generalizes to

$$\hat{p}(x) = \frac{1}{N(d_N(x))^d} \sum_{i=1}^N k\left(\frac{x - x_i}{d_N(x)}\right). \quad (16.10)$$

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

$$L_o(p, \omega_o) \approx \frac{1}{N_p d_N(p)^2} \sum_j^{N_p} k\left(\frac{p - p_j}{d_N(p)}\right) \beta_j f(p, \omega_o, \omega_j), \quad (16.11)$$

where we've switched to using N_p to denote the total number of emitted photons, the sum is over all of the photons, and scale factors for the photons are computed based on the density estimation Equation, (16.10). Because we know that the kernel function is zero for points farther away than the N th nearest neighbor distance $d_N(x)$, implementations of this sum only need to sum over the N closest neighbors.⁵

The error introduced by this interpolation can be difficult to quantify. Tracing more photons generally increases photon density and will almost always improve the results. When the photons are more closely spaced, it isn't necessary to use photons from as far away in the nearest-neighbor estimate. In general, the error at any given point will depend on how quickly the illumination is changing in its vicinity. 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. If traditional methods are used for direct lighting, then this is generally less of a problem since indirect illumination tends to be low frequency.

The original formulation of photon mapping was based on a two-pass algorithm where photons are first traced from the light sources. Photon interactions are recorded on surfaces of the scene and then organized in a spatial data structure (generally a kd-tree) for use at rendering time. A second pass follows paths starting from the camera; at each path vertex, nearby photons are used to estimate indirect illumination.

While this approach is effective, it has the limitation that the number of photons that can be used is limited by available memory since all of the photons must be stored. No

⁵ It should be noted that including the n -th photon of the nearest-neighbor search can introduce additional bias; see García et al. (2012) for a discussion including alternative estimators that avoid this issue.

improvement is possible once memory is full, even if one wants a higher quality result from using more photons. (In contrast, with path tracing, for example, one can always add more samples per pixel without any incremental storage cost—only the computational expense increases.)

Progressive Photon Mapping

The *progressive photon mapping* algorithm addressed this issue by restructuring the algorithm: first, a camera pass traces paths starting from the camera. Each pixel stores a representation of all of the non-specular path vertices found when generating paths that started in its extent. (For example, if a camera ray hits a diffuse surface, we might record the geometric information about the intersection point and the diffuse reflectance. If it hit a perfectly specular surface and then a diffuse surface, we would record the diffuse reflectance scaled by the specular BSDF value, and so forth.) We will dub these stored path vertices *visible points* in the following.⁶ A second pass traces photons from the light sources; at each photon–surface intersection, the photon contributes to the reflected radiance estimate for nearby visible points.

To understand how progressive photon mapping works, we consider a decomposition of the LTE into separate integrals over direct L_d and indirect L_i incident radiance at each vertex. (Recall the discussion of partitioning the LTE in Section 14.4.6.)

$$\begin{aligned} L(p, \omega_o) &= L_e(p, \omega_o) + \int_{S^2} f(p, \omega_o, \omega_i) L(p, \omega_i) |\cos \theta_i| d\omega_i \\ &= L_e(p, \omega_o) + \int_{S^2} f(p, \omega_o, \omega_i) L_d(p, \omega_i) |\cos \theta_i| d\omega_i \quad [16.12] \\ &\quad + \int_{S^2} f(p, \omega_o, \omega_i) L_i(p, \omega_i) |\cos \theta_i| d\omega_i. \end{aligned}$$

The emitted term is straightforward, and direct lighting can be handled using the usual approaches from Section 14.3.⁷ The indirect term with the integral over L_i is handled in one of two ways. First, a ray may be sampled from the BSDF’s sampling distribution and traced to find the next vertex in the path, where the same issue of how to estimate outgoing radiance is faced—just like a path tracer. Alternatively, the current point can be saved to receive illumination from photons. The final contribution of such a point to radiance at the film plane is found by summing the product of the photons’ weights and the path throughput weight of the sequence of vertices before it in the path.

For a perfectly specular BSDF, the only reasonable choice is to trace another ray: a photon will never arrive at exactly the right direction to match the delta distribution in the BSDF. For highly glossy surfaces, it’s also advisable to trace a ray, since it will take many photons for enough to hit a narrow specular lobe to compute an accurate estimate.

⁶ The term *hit points* is often used to describe these points in the photon mapping literature. We prefer the additional clarity of phrasing that includes the idea that these points are at least indirectly visible from the camera.

⁷ Other decompositions are also possible—for instance, some implementations use photon density estimation to handle the direct illumination component.

For diffuse surfaces, it's generally worth using photons, though it can be worthwhile to trace one last bounce. This approach is called *final gathering*; if photons are used only after a diffuse bounce, then any errors from insufficient photon density are generally less visible, though more camera paths may need to be traced to eliminate noise. (See Exercise 16.8 for further discussion of final gathering.)

With this approach, no photon storage is needed, and an arbitrary number of photons can be traced; the memory limit is instead tied to the storage required for the visible points and their reflection information. For high-resolution images or images that require many samples per pixel to resolve motion blur or depth of field, memory can still be a limitation.

Stochastic Progressive Photon Mapping

Stochastic progressive photon mapping is a modification of progressive photon mapping that doesn't suffer from either of these memory limits. Like progressive photon mapping, it generates a set of visible points from the camera but at a low sampling rate (e.g., following just one camera path per pixel). Next, a number of photons are shot from lights, accumulating contributions at nearby visible points. This process then repeats: the visible points are discarded, a new set is generated at different positions, another round of photons is traced, and so forth.

SPPM starts with the photon estimation equation, (16.11), and makes two adjustments. First, it uses a constant kernel function; in conjunction with the fact that the estimation is over 2D (the local tangent plane around the visible point), we have

$$L_o(p, \omega_o) \approx \frac{1}{N_p \pi r^2} \sum_j N_p \beta_j f(p, \omega_o, \omega_j),$$

where, as before, N_p is the number of photons emitted from the light sources, and πr^2 is the surface area of the disk-shaped kernel function.

The second adjustment, based on an approach first implemented in progressive photon mapping, is to progressively reduce the photon search radius as more photons contribute to the visible point. The general idea is that as more photons that are found within the search radius, we have more evidence that a sufficient density of photons is arriving to estimate the incident radiance distribution well. By reducing the radius, we ensure that future photons that are used will be closer to the point and thus contribute to a more accurate estimate of the incident radiance distribution.

Reducing the radius requires an adjustment to how the reflected radiance estimate is computed, as now the photons in the sum in Equation (16.11) come from different radii. The following three update rules describe how to update the radius and dependent variables:

$$\begin{aligned} N_{i+1} &= N_i + \gamma M_i \\ r_{i+1} &= r_i \sqrt{\frac{N_{i+1}}{N_i + M_i}} \\ \tau_{i+1} &= (\tau_i + \Phi_i) \frac{r_{i+1}^2}{r_i^2}, \end{aligned} \tag{16.13}$$

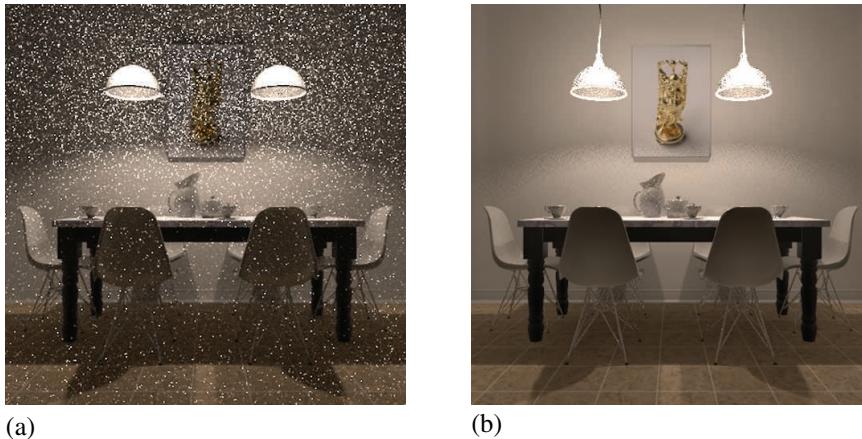


Figure 16.8: Scene rendered with (a) path tracing and (b) stochastic progressive photon mapping, using approximately the same amount of computation. In this case, photon mapping is effective at handling light paths from the light source through the glass light fixtures, while path tracing gives a result with high variance. (*Scene courtesy "Wig42" from blendswap.com*)

where N_i is the number of photons that have contributed to the point after the i th iteration, M_i is the number of photons that contributed during the current iteration, r_i is the search radius to use for the i th iteration, τ maintains the sum of products of photons with BSDF values, and Φ_i is computed during the i th iteration as

$$\Phi_i = \sum_j^{M_i} \beta_j f(\mathbf{p}, \omega_o, \omega_j). \quad [16.14]$$

The γ parameter, which is typically around $2/3$, determines how quickly the contributions from photons from earlier iterations, with wider search radii, are faded out. (Hachisuka and Jensen's original paper on SPPM (2009) uses the notation α for this quantity; we opt for γ here, having used α for other quantities already.)

Note that the radius is a per-pixel property, not a per-visible-point property. Remarkably, a consistent estimate for the reflected radiance is computed even with this single radius shared over all of the series of visible points in the pixel. We won't show this derivation here, but with the rules in Equation (16.13), as the number of photons traced $N_p \rightarrow \infty$, $r \rightarrow \infty$, and the reflected radiance estimates are consistent and converge to the correct values.

Figure 16.8 shows a rendered scene with path tracing and SPPM. SPPM is much more effective at handling light that passes through the glass light fixtures than the path tracing algorithm is.

16.2.3 SPPMIntegrator

The `SPPMIntegrator`, implemented in the files `integrators/sppm.h` and `integrators/sppm.cpp`, implements the SPPM light transport algorithm.

```
<SPPM Declarations> ≡
    class SPPMIntegrator : public Integrator {
public:
    <SPPMIntegrator Public Methods>
private:
    <SPPMIntegrator Private Data 973>
};
```

The SPPMIntegrator constructor isn't particularly interesting; it just sets various member variables with values passed in. We therefore won't include it here but will discuss the various member variables that configure the SPPMIntegrator's operation as they appear in the following.

The SPPMIntegrator is not a SamplerIntegrator, so it implements its own Render() method. After some initial setup has been performed, it runs a number of iterations of the SPPM algorithm, finding a set of visible points and then accumulating illumination from photons at them. Each iteration creates a new path starting from the camera in each pixel, which helps with antialiasing geometric edges and sampling motion blur and depth of field well.

```
<SPPM Method Definitions> ≡
void SPPMIntegrator::Render(const Scene &scene) {
    <Initialize pixelBounds and pixels array for SPPM 973>
    <Compute lightDistr for sampling lights proportional to power 974>
    <Perform nIterations of SPPM integration 975>
}
```

The pixels array stores a SPPMPixel (to be defined shortly) for each pixel in the final image.

```
<Initialize pixelBounds and pixels array for SPPM> ≡ 973
    Bounds2i pixelBounds = camera->film->croppedPixelBounds;
    int nPixels = pixelBounds.Area();
    std::unique_ptr<SPPMPixel[]> pixels(new SPPMPixel[nPixels]);
    for (int i = 0; i < nPixels; ++i)
        pixels[i].radius = initialSearchRadius;
```

Bounds2i 76
 Camera 356
 Camera::film 356
 Film::croppedPixelBounds 485
 Float 1062
 Integrator 25
 SamplerIntegrator 25
 Scene 23
 SPPMIntegrator::camera 973
 SPPMIntegrator::
 initialSearchRadius
 973
 SPPMPixel 974
 SPPMPixel::radius 974

A user-supplied radius, initialSearchRadius, is used for r_0 , the initial search radius for photons. If the supplied radius is too large, too many photons will contribute to visible points during early iterations (before the radius is automatically decreased), which may be inefficient. If it's too small, not enough photons will be found to estimate incident radiance well. A radius corresponding to a few pixels in the final image is generally a good starting point.

```
<SPPMIntegrator Private Data> ≡ 973
    std::shared_ptr<const Camera> camera;
    const Float initialSearchRadius;
```

The SPPMPixel structure serves three purposes. First, it stores the current estimated average radiance visible over the extent of a pixel (including the time the shutter is open

and accounting for depth of field, if present). Second, it stores parameters related to the photon density estimation for the pixel (e.g., various quantities from Equation (16.13)). Finally, it stores the geometric and reflection information for a visible point in the pixel after the camera pass.

(SPPM Local Definitions) \equiv

```
struct SPPMPixel {
    (SPPMPixel Public Methods)
    (SPPMPixel Public Data 974)
};
```

(SPPMPixel Public Data) \equiv

974

```
Float radius = 0;
```

Working with weighted photons allows for two very different overall sampling approaches: on the one hand, we could try to distribute photons uniformly with a weight that approximates the irradiance on surfaces, while also accounting for discontinuities and other important geometric and lighting features. However, this type of photon distribution is challenging to realize for general input; hence we will instead strive to generate photons have the same (or similar) weights, so that it is their varying density throughout the scene that represents the variation in illumination.

Furthermore, if the photon weights on a surface have substantial variation that is not related to the irradiance, there can be unpleasant image artifacts: if one photon takes on a much larger weight than the others, a bright circular artifact will reveal the region of the scene where that photon contributes to the interpolated radiance.

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 photon path from is chosen according to a PDF defined by the lights' respective powers. 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 the same number of photons from all, with larger weights for the more powerful lights.

(Compute lightDistr for sampling lights proportional to power) \equiv

973

```
std::unique_ptr<Distribution1D> lightDistr =
    ComputeLightPowerDistribution(scene);
```

(Integrator Utility Functions) $+ \equiv$

```
std::unique_ptr<Distribution1D> ComputeLightPowerDistribution(
    const Scene &scene) {
    std::vector<Float> lightPower;
    for (const auto &light : scene.lights)
        lightPower.push_back(light->Power().y());
    return std::unique_ptr<Distribution1D>(
        new Distribution1D(&lightPower[0], lightPower.size()));
}
```

```
ComputeLightPowerDistribution()
974
Distribution1D 758
Float 1062
Light::Power() 717
Scene 23
Scene::lights 23
Spectrum::y() 325
```

Each iteration of the SPPM algorithm traces a new path from the camera at each pixel and then collects incident photons at each path's endpoint. Figure 16.9 shows the effect of increasing the number of iterations with SPPM. Here, we see the caustic from light

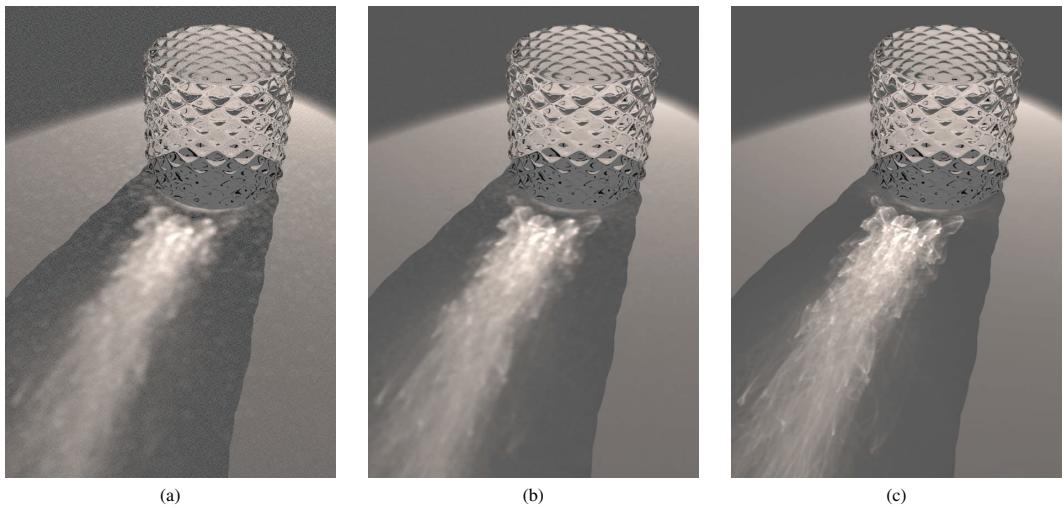


Figure 16.9: Effect of the number of iterations on results from the SPPMIntegrator. As the number of iterations increases (and thus, the more photons are traced), the quality of the final result improves. Note in particular how the size of the visible circular blotches gets smaller. (a) 10 iterations, (b) 100 iterations, (c) 10,000 iterations. (Model courtesy Simon Wendsche.)

passing through the glass becoming increasingly sharper. In general, more iterations improve the sampling of visible edges, motion blur, and depth of field, but as more photons are accumulated in each pixel, the indirect lighting estimate becomes more accurate. Note how the artifacts from under-sampling here are low-frequency blotches, a quite different visual artifact from the high-frequency noise from under-sampling seen in path tracing.

The implementation here uses a `HaltonSampler` to generate camera paths; doing so ensures that well-distributed samples are used over all of the iterations in the aggregate.

```

⟨Perform nIterations of SPPM integration⟩ ≡
    HaltonSampler sampler(nIterations, pixelBounds);
    ⟨Compute number of tiles to use for SPPM camera pass 976⟩
    for (int iter = 0; iter < nIterations; ++iter) {
        ⟨Generate SPPM visible points 976⟩
        ⟨Trace photons and accumulate contributions 983⟩
        ⟨Update pixel values from this pass's photons 989⟩
        ⟨Periodically store SPPM image in film and write image⟩
    }

```

```

HaltonSampler 450
SamplerIntegrator 25
SPPMIntegrator::nIterations
975

```

973

```

⟨SPPMIntegrator Private Data⟩ +≡
    const int nIterations;

```

973

16.2.4 ACCUMULATING VISIBLE POINTS

Similar to the `SamplerIntegrator`, the `SPPMIntegrator` decomposes the image into tiles of 16 pixels square and parallelizes the generation of camera paths and visible points

over these tiles. The number of tiles is computed in the same manner as in the fragment (*Compute number of tiles, nTiles, to use for parallel rendering*).

(Compute number of tiles to use for SPPM camera pass) ≡

975

```
Vector2i pixelExtent = pixelBounds.Diagonal();
const int tileSize = 16;
Point2i nTiles((pixelExtent.x + tileSize - 1) / tileSize,
                (pixelExtent.y + tileSize - 1) / tileSize);
```

Unlike the path tracer, where the BSDF can be discarded after estimating the direct illumination and sampling the outgoing direction at each vertex, here we need to store the BSDFs for the visible points until the photon pass for the current iteration is done. Therefore, the MemoryArenas used for allocating the BSDFs during camera path tracing aren't reset at the end of loop iterations here.

Note also that we only allocate one arena per worker thread used to run the parallel for loop and use the ThreadIndex global to index into the vector, rather than allocating a separate arena for each loop iteration. In this way, we avoid the overhead of having many separate MemoryArenas while still ensuring that each arena is not used by more than one processing thread (which would lead to race conditions in the MemoryArena methods).

(Generate SPPM visible points) ≡

975

```
std::vector<MemoryArena> perThreadArenas(MaxThreadIndex());
ParallelFor2D(
    [&](Point2i tile) {
        MemoryArena &arena = perThreadArenas[ThreadIndex];
        (Follow camera paths for tile in image for SPPM 976)
    }, nTiles);
(Create grid of all SPPM visible points 979)
```

We also need a unique Sampler for the thread processing the tile; as before, Sampler::Clone() provides one.

(Follow camera paths for tile in image for SPPM) ≡

976

```
int tileIndex = tile.y * nTiles.x + tile.x;
std::unique_ptr<Sampler> tileSampler = sampler.Clone(tileIndex);
(Compute tileBounds for SPPM tile)
for (Point2i pPixel : tileBounds) {
    (Prepare tileSampler for pPixel 977)
    (Generate camera ray for pixel for SPPM 977)
    (Follow camera ray path until a visible point is created 977)
}
```

The fragment *(Compute tileBounds for SPPM tile)* is very similar to *(Compute sample bounds for tile)* from Section 1.3.4 and therefore won't be included here.

Recall that in SamplerIntegrators, the sample vectors in each pixel are all requested in sequence until the last one is consumed, at which point work starts on the next pixel. In contrast, the first SPPM iteration uses the first sample vector for each pixel; later, the second iteration uses the second sample vector, and so forth. (In other words, the loop nesting has been interchanged from "for each pixel, for each sample number," to "for

Bounds2::Diagonal() 80
 BSDF 572
 MaxThreadIndex() 1093
 MemoryArena 1074
 ParallelFor2D() 1093
 Point2i 68
 Sampler 421
 Sampler::Clone() 424
 ThreadIndex 1089
 Vector2i 60

each sample number, for each pixel.”) It is for just this use case that the Sampler provides the `SetSampleNumber()` method, which configures the sampler to provide samples from the given sample vector for the pixel.

```
{Prepare tileSampler for pPixel} ≡ 976
    tileSampler->StartPixel(pPixel);
    tileSampler->SetSampleNumber(iter);
```

We can now start the path from the camera, following the usual approach. As with the `PathIntegrator`, the `beta` variable holds the current path throughput weight β .

```
{Generate camera ray for pixel for SPPM} ≡ 976
    CameraSample cameraSample = tileSampler->GetCameraSample(pPixel);
    RayDifferential ray;
    Spectrum beta = camera->GenerateRayDifferential(cameraSample, &ray);
```

Path tracing can now proceed. As with most other `Integrators`, the path length is limited by a predefined maximum depth.

```
{Follow camera ray path until a visible point is created} ≡ 976
    (Get SPPMPixel for pPixel 977)
    bool specularBounce = false;
    for (int depth = 0; depth < maxDepth; ++depth) {
        SurfaceInteraction isect;
        if (!scene.Intersect(ray, &isect)) {
            (Accumulate light contributions for ray with no intersection 977)
            break;
        }
        (Process SPPM camera ray intersection 978)
    }

Camera::
    GenerateRayDifferential() 357
    CameraSample 357
Light::Le() 741
PathIntegrator 875
Point2i 68
RayDifferential 75
Sampler::GetCameraSample() 423
Sampler::SetSampleNumber() 424
Sampler::StartPixel() 422
Scene::Intersect() 24
Scene::lights 23
Spectrum 315
SPPMIntegrator::maxDepth 977
SPPMPixel 974
SPPMPixel::Ld 978
SurfaceInteraction 116
```

To find the `SPPMPixel` in the `pixels` array for the current pixel, we need to offset by the minimum pixel coordinate and convert to a linear index.

```
{Get SPPMPixel for pPixel} ≡ 977
    Point2i pPixel0 = Point2i(pPixel - pixelBounds.pMin);
    int pixelOffset = pPixel0.x +
                      pPixel0.y * (pixelBounds.pMax.x - pixelBounds.pMin.x);
    SPPMPixel &pixel = pixels[pixelOffset];
```

As described in Section 16.2.2, a regular direct lighting calculation is performed at each vertex of the camera path. Thus, for rays that don’t intersect any scene geometry, infinite area lights must be allowed a chance to contribute direct lighting via the `Light::Le()` method.

```
{Accumulate light contributions for ray with no intersection} ≡ 977
    for (const auto &light : scene.lights)
        pixel.Ld += beta * light->Le(ray);
```

`SPPMPixel::Ld` records the weighted sum of emitted and reflected direct illumination for all camera path vertices for the pixel (in other words, the first two terms of Equation (16.12)). Note that these terms are also evaluated at vertices found by sampling the BSDF and tracing a ray for the third term; `Ld` doesn't just store direct illumination at the first vertex. Because this sum of outgoing radiance includes contributions of all of the samples in a pixel, this value must be divided by `SPPMIntegrator::nIterations` to get the average direct illumination for the final pixel radiance estimate.

(SPPMPixel Public Data) +≡

974

`Spectrum Ld;`

More commonly, the ray intersects a surface and *(Process SPPM camera ray intersection)* executes

(Process SPPM camera ray intersection) ≡

977

(Compute BSDF at SPPM camera ray intersection 978)

(Accumulate direct illumination at SPPM camera ray intersection 978)

(Possibly create visible point and end camera path 979)

(Spawn ray from SPPM camera path vertex)

First, we need the BSDF at the intersection point. Recall from Section 11.3 that a `nullptr`-valued `BSDF *` means that an intersection should be ignored, as the surface intersected is only in the scene to delineate the boundary of participating media. The `SPPMIntegrator` does not account for participating media; thus we simply skip over the intersection and restart the current loop iteration.

(Compute BSDF at SPPM camera ray intersection) ≡

978

```
isect.ComputeScatteringFunctions(ray, arena, true);
if (!isect.bsdf) {
    ray = isect.SpawnRay(ray.d);
    --depth;
    continue;
}
const BSDF &bsdf = *isect.bsdf;
```

As in other integrators, emitted radiance from the surface is only included for the first intersection from the camera or after a specular bounce, where no direct lighting calculation was possible.

(Accumulate direct illumination at SPPM camera ray intersection) ≡

978

```
Vector3f wo = -ray.d;
if (depth == 0 || specularBounce)
    pixel.Ld += beta * isect.Le(wo);
pixel.Ld += beta *
    UniformSampleOneLight(isect, scene, arena, *tileSampler);
```

The implementation here creates a visible point at the first diffuse surface found or if the path is about to hit its maximum length and we have a glossy surface. As explained earlier, a visible point at a perfectly specular surface will never respond to incident photons, and a glossy surface may have high variance if the specular lobe is tight and not enough photons have arrived to represent the incident radiance distribution well.

BSDF 572

Ray::d 73

Spectrum 315

SPPMIntegrator 973

SPPMIntegrator::nIterations
975

SPPMPixel::Ld 978

SurfaceInteraction::bsdf 250

SurfaceInteraction::
ComputeScatteringFunctions()
578

SurfaceInteraction::Le() 734

UniformSampleOneLight() 856

Vector3f 60

```
<Possibly create visible point and end camera path> ≡ 978
    bool isDiffuse =
        bsdf.NumComponents(BxDFType(BSDF_DIFFUSE | BSDF_REFLECTION |
                                     BSDF_TRANSMISSION)) > 0;
    bool isGlossy =
        bsdf.NumComponents(BxDFType(BSDF_GLOSSY | BSDF_REFLECTION |
                                     BSDF_TRANSMISSION)) > 0;
    if (isDiffuse || (isGlossy && depth == maxDepth - 1)) {
        pixel.vp = {isect.p, wo, &bsdf, beta};
        break;
    }
```

The `VisiblePoint` structure records a point found along a camera path at which we'll look for nearby photons during the photon shooting path. It stores enough information to compute the reflected radiance due to an incident photon, which in turn is scaled by the path throughput weight of the visible point to find the overall contribution to the original image sample.

```
<SPPMPixel Public Data> +≡ 974
    struct VisiblePoint {
        <VisiblePoint Public Methods>
        Point3f p;
        Vector3f wo;
        const BSDF *bsdf = nullptr;
        Spectrum beta;
    } vp;
```

Sampling a ray leaving a vertex follows the same form as *(Sample BSDF to get new path direction)* in the `PathIntegrator` and is therefore not included here.

16.2.5 VISIBLE POINT GRID CONSTRUCTION

During the photon pass, whenever a photon intersects a surface, we need to efficiently find the visible points where the distance from the visible point to the photon's intersection is less than the current search radius r_i for the visible point's pixel. The implementation here uses a uniform grid over the bounding box of all of the visible points; Exercise 16.7 at the end of the chapter has you implement other data structures in place of the grid and investigate trade-offs.

```
<Create grid of all SPPM visible points> ≡ 976
    <Allocate grid for SPPM visible points 980>
    <Compute grid bounds for SPPM visible points 981>
    <Compute resolution of SPPM grid in each dimension 981>
    <Add visible points to SPPM grid 981>
```

The grid is usually sparsely filled; many voxels have no visible points inside their extent. (For starters, any voxel in the grid with no surfaces in its volume will never have a visible point in it.) Therefore, rather than allocating storage for all of the voxels, the grid is represented by a hash table where a hash function transforms 3D voxel coordinates to an index into the grid array.

`BSDF` 572

`BSDF::NumComponents()` 573

`BSDF_DIFFUSE` 513

`BSDF_GLOSSY` 513

`BSDF_REFLECTION` 513

`BSDF_TRANSMISSION` 513

`BxDFType` 513

`Interaction::p` 115

`PathIntegrator` 875

`Point3f` 68

`Spectrum` 315

`SPPMPixel::vp` 979

`Vector3f` 60

In the following, we'll construct the grid in parallel using multiple threads. These threads will use atomic operations to update the grid, so a `std::atomic` is used for the grid voxel element type here.

(Allocate grid for SPPM visible points) \equiv

979

```
int hashSize = nPixels;
std::vector<std::atomic<SPPMPixelListNode *>> grid(hashSize);
```

Each grid cell stores a linked list, where each node points to an `SPPMPixel` whose visible point's search volume overlaps the grid cell. A visible point may overlap multiple grid cells, so it's desirable to keep the node representation compact by only storing a pointer to the `SPPMPixel` rather than making a copy for each cell it overlaps.

(SPPM Local Definitions) $+ \equiv$

```
struct SPPMPixelListNode {
    SPPMPixel *pixel;
    SPPMPixelListNode *next;
};
```

If there's no visible point for the pixel for the current iteration, the pixel will have a path throughput weight $\beta = 0$ (and no attempt should be made to place a visible point in the grid for that pixel). This case can happen if the path from the camera leaves the scene or is terminated before intersecting a diffuse surface.

Otherwise, the implementation here computes the bounding box centered at the visible point with extent $\pm r_i$, the current photon search radius for the pixel. In turn, when we later have a photon intersection point, we will only need to consider the visible points for the grid cell that the photon is in to find the visible points that the photon may contribute to (Figure 16.10). Because different visible points will have different search radii depending on how many photons have contributed to their pixel so far, it would otherwise be unwieldy to find the potentially relevant visible points for a photon if the

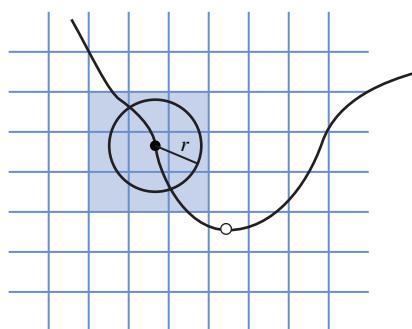


Figure 16.10: Given a visible point (filled circle) with search radius r , the visible point is added to the linked list in all grid cells that the bounding box of the sphere of radius r overlaps. Given a photon incident on a surface in the scene (open circle), we only need to check the visible points in the voxel the photon is in to find the ones that it may contribute to.

`SPPMPixel` 974

`SPPMPixelListNode` 980

visible points were stored without accounting for the volume of space they will accept photons from.

```
(Compute grid bounds for SPPM visible points) ≡ 979
    Bounds3f gridBounds;
    Float maxRadius = 0.;
    for (int i = 0; i < nPixels; ++i) {
        const SPPMPixel &pixel = pixels[i];
        if (pixel.vp.beta.IsBlack())
            continue;
        Bounds3f vpBound = Expand(Bounds3f(pixel.vp.p), pixel.radius);
        gridBounds = Union(gridBounds, vpBound);
        maxRadius = std::max(maxRadius, pixel.radius);
    }
```

Given the overall bound of the grid, we need to decide how large the voxels should be and thus how finely to subdivide space. On one hand, if the voxels are too large, the photon shooting pass will be inefficient, as each photon will have to check many visible points to see if it contributes to each one. If they're too small, then each visible point will overlap many voxels, and the memory needed to represent the grid will be excessive.

Here, we compute an initial resolution such that the voxel width in the largest grid dimension is roughly equal to the largest current search radius of all of the visible points. This limits the maximum number of voxels any visible point can overlap. In turn, resolutions for the other two dimensions of the grid are set so that voxels have roughly the same width in all dimensions.

```
(Compute resolution of SPPM grid in each dimension) ≡ 979
    Vector3f diag = gridBounds.Diagonal();
    Float maxDiag = MaxComponent(diag);
    int baseGridRes = (int)(maxDiag / maxRadius);
    int gridRes[3];
    for (int i = 0; i < 3; ++i)
        gridRes[i] = std::max((int)(baseGridRes * diag[i] / maxDiag), 1);

Bounds3::Diagonal() 80
Bounds3::Expand() 79
Bounds3::Union() 78
Bounds3f 76
Float 1062
MemoryArena 1074
ParallelFor() 1088
Spectrum::IsBlack() 317
SPPMPixel 974
SPPMPixel::radius 974
SPPMPixel::vp 979
SPPMPixelListNode 980
ThreadIndex 1089
Vector3::MaxComponent() 66
Vector3f 60
VisiblePoint::beta 979
```

The visible points can now be added to the grid. Because both the grid and the BSDFs for visible points from the camera path must be kept around until the end of the photon tracing pass, we can reuse the per-thread MemoryArenas that were created earlier for the BSDFs to allocate SPPMPixelListNode nodes.

```
(Add visible points to SPPM grid) ≡ 979
    ParallelFor(
        [&](int pixelIndex) {
            MemoryArena &arena = perThreadArenas[ThreadIndex];
            SPPMPixel &pixel = pixels[pixelIndex];
            if (!pixel.vp.beta.IsBlack()) {
                (Add pixel's visible point to applicable grid cells 982)
            }
        }, nPixels, 4096);
```

Each visible point is added to all of the grid cells that its bounding box overlaps.

(Add pixel's visible point to applicable grid cells) ≡

981

```
Float radius = pixel.radius;
Point3i pMin, pMax;
ToGrid(pixel.vp.p - Vector3f(radius, radius, radius),
       gridBounds, gridRes, &pMin);
ToGrid(pixel.vp.p + Vector3f(radius, radius, radius),
       gridBounds, gridRes, &pMax);
for (int z = pMin.z; z <= pMax.z; ++z)
    for (int y = pMin.y; y <= pMax.y; ++y)
        for (int x = pMin.x; x <= pMax.x; ++x) {
            (Add visible point to grid cell (x, y, z) 982)
        }
}
```

ToGrid() returns the coordinates of the voxel in the grid that the given point lies in. The Boolean return value indicates whether the point p is inside the grid's bounds; if it isn't, the returned coordinates pi are clamped to be inside the range of valid coordinates.

(SPPM Local Definitions) +≡

```
static bool ToGrid(const Point3f &p, const Bounds3f &bounds,
                   const int gridRes[3], Point3i *pi) {
    bool inBounds = true;
    Vector3f pg = bounds.Offset(p);
    for (int i = 0; i < 3; ++i) {
        (*pi)[i] = (int)(gridRes[i] * pg[i]);
        inBounds &= ((*pi)[i] >= 0 && (*pi)[i] < gridRes[i]);
        (*pi)[i] = Clamp((*pi)[i], 0, gridRes[i] - 1);
    }
    return inBounds;
}
```

hash() hashes the coordinates of the voxel, returning an index into the grid array defined earlier; it is a straightforward hash function, and its implementation isn't included here.⁸ A new SPPMPixelListNode is allocated and the task now is to add this list node to the head of the linked list in grid[h].

(Add visible point to grid cell (x, y, z)) ≡

982

```
int h = hash(Point3i(x, y, z), hashSize);
SPPMPixelListNode *node = arena.Alloc<SPPMPixelListNode>();
node->pixel = &pixel;
(Atomically add node to the start of grid[h]'s linked list 983)
```

Given the grid index, atomic operations can be used to allow multiple threads to add visible points to the grid concurrently without needing to hold any locks. (See Appen-

Bounds3::Offset() 81
 Bounds3f 76
 Clamp() 1062
 Float 1062
 hash() 982
 Point3f 68
 Point3i 68
 SPPMPixel::radius 974
 SPPMPixel::vp 979
 SPPMPixelListNode 980
 SPPMPixelListNode::pixel 980
 ToGrid() 982
 Vector3f 60
 VisiblePoint::p 979

⁸ Note that hash collisions are possible—different cells in the grid may hash to the same index. This is almost fine: photons will end up checking more visible points than they need to, but the test with the pixel's search radius will reject the visible points that are too far away anyway. See, however, Exercise 16.9 for one nit related to collisions.

dix A.6.2 for further discussion of atomic operations.) In the absence of concurrency, we'd just want to set `node->next` to point to the head of the list in `grid[h]` and assign `node` to `grid[h]`. That approach will not work if multiple threads are updating the lists concurrently; it's possible that the first assignment will become stale: another thread might modify `grid[h]` between the current thread initializing `node->next` and then trying to assign `node` to `grid[h]`.

The `compare_exchange_weak()` method of `std::atomic` addresses this issue: the first parameter is the value that we expect `grid[h]` to have, and the second gives the value that we'd like to set it to. It performs the assignment only if our expectation was correct. Thus, in the common case, `node->next` and `grid[h]` have the same pointer value, the assignment occurs, and `true` is returned. The node has been added to the list.

If the pointer stored in `grid[h]` has been changed by another thread, then `compare_exchange_weak()` actually updates the first parameter, `node->next`, with the current value of `grid[h]` before returning `false`. We are thus all set to try the atomic compare and exchange again, as `node->next` points to the new head of the list. This process continues until the assignment is successful.

(Atomically add node to the start of grid[h]'s linked list) ≡

982

```
node->next = grid[h];
while (grid[h].compare_exchange_weak(node->next, node) == false)
    ;
```

16.2.6 ACCUMULATING PHOTON CONTRIBUTIONS

Given the grid of visible points, the `SPPMIntegrator` can now follow photon paths through the scene. The total of `photonsPerIteration` photons to be traced for the current iteration are traced using multiple threads. A separate `MemoryArena` is available for each worker thread; this arena is reset after each photon path, so a new pool of per-thread arenas is allocated rather than reusing the one used for BSDFs and grid linked list nodes.

(Trace photons and accumulate contributions) ≡

975

```
std::vector<MemoryArena> photonShootArenas(MaxThreadIndex());
ParallelFor(
    [&](int photonIndex) {
        MemoryArena &arena = photonShootArenas[ThreadIndex];
        (Follow photon path for photonIndex 984)
        arena.Reset();
    }, photonsPerIteration, 8192);
```

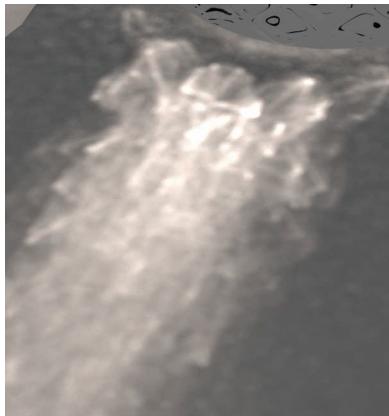
MaxThreadIndex() 1093
 MemoryArena 1074
 MemoryArena::Reset() 1076
 ParallelFor() 1088
`SPPMIntegrator::`
`photonsPerIteration` 983
`SPPMPixelListNode::next` 980
 ThreadIndex 1089

There's a balance to strike in choosing the number of photons to trace for each SPPM iteration: too many, and pixels' radii won't have a chance to decrease as more photons arrive and too many too-far-away photons are used. Too few, and the overhead of finding visible points and making the grid of them won't be amortized over enough photons. In practice, a few hundred thousand to a few million per iteration generally works well (see Figure 16.11).

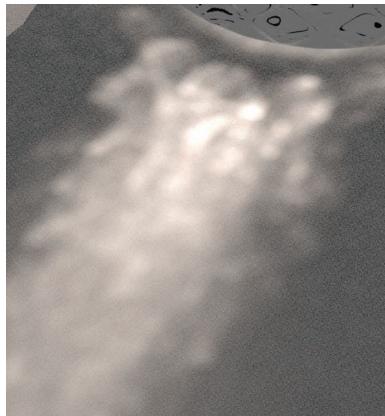
(SPPMIntegrator Private Data) +≡

973

```
const int photonsPerIteration;
```



(a)



(b)

Figure 16.11: The Effect of Varying the Number of Photons Traced Per Iteration. The number of iterations is set so that the total number of photons traced is the same—10 million—for both cases. (a) 10,000 photons (1000 iterations): results are good, and rendering time is 137 seconds; time spent creating visible points is 68% of the total. (b) 1,000,000 photons (10 iterations): results are much blurrier, though rendering time has dropped to 50 seconds, most of it due to spending much less time on the camera pass.

A Halton sequence provides a set of well-distributed sample points for all of the photon paths over all of the iterations. `haltonIndex` records the index of the Halton sequence (corresponding to a in Equation (7.7)) for the current photon; it can also be seen as a global index of photons traced. In other words, it starts at 0 for the first photon and passes through all subsequent integer values for following photons. It's important to use a 64-bit integer for this value, since a 32-bit `int` would overflow after roughly 2 billion photons; many more photons may be needed for high-quality images.

The dimension of the sample, corresponding to the b th prime number in Equation (7.7), is maintained in `haltonDim`.

```
(Follow photon path for photonIndex) ≡ 983
    uint64_t haltonIndex = (uint64_t)iter * (uint64_t)photonsPerIteration +
                           photonIndex;
    int haltonDim = 0;
    (Choose light to shoot photon from 985)
    (Compute sample values for photon ray leaving light source 985)
    (Generate photonRay from light source and initialize beta 985)
    (Follow photon path through scene and record intersections 986)
```

Which light to start the path from is determined by sampling from the PDF based on light power computed previously. The first dimension of the Halton sequence is used for the sample value.

```
SPPMIntegrator::  
photonsPerIteration  
983
```

```
(Choose light to shoot photon from) ≡ 984
    Float lightPdf;
    Float lightSample = RadicalInverse(haltonDim++, haltonIndex);
    int lightNum = lightDistr->SampleDiscrete(lightSample, &lightPdf);
    const std::shared_ptr<Light> &light = scene.lights[lightNum];
```

The next five dimensions of the sample from the Halton sequence are used for the sample values used to generate the ray leaving the light source.

```
(Compute sample values for photon ray leaving light source) ≡ 984
    Point2f uLight0(RadicalInverse(haltonDim, haltonIndex),
                    RadicalInverse(haltonDim + 1, haltonIndex));
    Point2f uLight1(RadicalInverse(haltonDim + 2, haltonIndex),
                    RadicalInverse(haltonDim + 3, haltonIndex));
    Float uLightTime = Lerp(RadicalInverse(haltonDim + 4, haltonIndex),
                           camera->shutterOpen, camera->shutterClose);
    haltonDim += 5;
```

After the light has been chosen, its `Sample_Le()` method is used to sample an outgoing ray. Given a light, a ray from the light source is sampled and its β value is initialized based on Equation (16.8):

$$\beta = \frac{|\cos \omega_0| L_e(p_0, \omega_0)}{p(\text{light}) p(p_0, \omega_0)}, \quad (16.15)$$

where $p(\text{light})$ is the probability for sampling this particular light and $p(p_0, \omega_0)$ is the product of the area and directional densities for sampling this particular ray leaving the light. Intersecting this ray against the scene geometry to obtain p_1 also samples part of the geometric term $G(p_0 \leftrightarrow p_1)$ except for a cosine factor that must be explicitly integrated into the particle weight β .

```
(Generate photonRay from light source and initialize beta) ≡ 984
AbsDot() 64
Camera::shutterClose 356
Camera::shutterOpen 356
Distribution1D::
    SampleDiscrete()
    760
Float 1062
Lerp() 1079
Light 714
Light::Sample_Le() 955
Normal3f 71
Point2f 68
RadicalInverse() 444
RayDifferential 75
Scene::lights 23
Spectrum 315
Spectrum::IsBlack() 317
```

Now the integrator can start following the path through the scene, updating β after each scattering event. The photon makes no contribution at the first intersection found after it left the light source, since that intersection represents direct illumination, which was already accounted for when tracing paths starting from the camera. For subsequent intersections, illumination is contributed to nearby visible points.

(Follow photon path through scene and record intersections) ≡ 984

```
SurfaceInteraction isect;
for (int depth = 0; depth < maxDepth; ++depth) {
    if (!scene.Intersect(photonRay, &isect))
        break;
    if (depth > 0) {
        <Add photon contribution to nearby visible points 986>
    }
    <Sample new photon ray direction 987>
}
```

Given a photon intersection, `ToGrid()`'s return value indicates if it's within the extent of the grid. If it isn't, then by construction, none of the visible points is interested in this photon's contribution. Otherwise, the visible points in the grid cell all need to be checked to see if the photon is within their radius.

(Add photon contribution to nearby visible points) ≡ 986

```
Point3i photonGridIndex;
if (ToGrid(isect.p, gridBounds, gridRes, &photonGridIndex)) {
    int h = hash(photonGridIndex, hashSize);
    <Add photon contribution to visible points in grid[h] 986>
}
```

Recall that `grid` stores `std::atomic` pointers to `SPPMPixelListNodes`. Normally, reading from a `std::atomic` value means that the compiler must be careful to not reorder instructions that read or write memory around the read of the value of `grid[h]`; this constraint is necessary so that lock-free algorithms will work as expected. In this case, the grid has been constructed and no other threads are concurrently modifying it. Therefore, it's worthwhile to use the `std::atomic.load()` method and letting it know that the "relaxed" memory model, which doesn't have these constraints, can be used to read the initial grid pointer. This approach has a significant performance benefit: for a simple scene of a few hundred triangles (where not too much time is spent tracing rays), the photon pass runs in 20% less time using this memory model on a 2015-era CPU.

(Add photon contribution to visible points in grid[h]) ≡ 986

```
for (SPPMPixelListNode *node = grid[h].load(std::memory_order_relaxed);
     node != nullptr; node = node->next) {
    SPPMPixel &pixel = *node->pixel;
    Float radius = pixel.radius;
    if (DistanceSquared(pixel.vp.p, isect.p) > radius * radius)
        continue;
    <Update pixel Φ and M for nearby photon 987>
}
```

Given a photon contribution, we need to update the sum for the pixel's scattered radiance estimate from Equation (16.14). The total number of contributing photons in this pass is stored in `M`, and the sum of the product of BSDF values with particle weights is stored in `Phi`.

```
DistanceSquared() 70
Float 1062
hash() 982
Interaction::p 115
Point3i 68
Scene::Intersect() 24
SPPMIntegrator::maxDepth 977
SPPMPixel 974
SPPMPixel::radius 974
SPPMPixel::vp 979
SPPMPixelListNode 980
SPPMPixelListNode::next 980
SPPMPixelListNode::pixel 980
SurfaceInteraction 116
ToGrid() 982
VisiblePoint::p 979
```

```
{Update pixel  $\Phi$  and  $M$  for nearby photon} ≡ 986
    Vector3f wi = -photonRay.d;
    Spectrum Phi = beta * pixel.vp.bsdf->f(pixel.vp.wo, wi);
    for (int i = 0; i < Spectrum::nSamples; ++i)
        pixel.Phi[i].Add(Phi[i]);
    ++pixel.M;
```

Each pixel's Φ and M values are stored using atomic variables, which in turn allows multiple threads to safely concurrently update their values. Because pbrt's `Spectrum` class doesn't allow atomic updates, Φ is instead represented with an array of `AtomicFloat` coefficients for each spectral sample. This representation will require some manual copying of values to and from Φ to `Spectrum`-typed variables in the following, but we think that this small awkwardness is preferable to the complexity of, for example, a new `AtomicSpectrum` type.

```
{SPPMPixel Public Data} +≡ 974
    AtomicFloat Phi[Spectrum::nSamples];
    std::atomic<int> M;
```

Having recorded the photon's contribution, the integrator needs to choose a new outgoing direction from the intersection point and update the β value to account for the effect of scattering. Equation (16.7) shows how to incrementally update the particle weight after a scattering event: given some weight $\beta_{i,j}$ that represents the weight for the j th intersection of the i th particle history, after a scattering event where a new vertex $p_{i,j+1}$ has been sampled, the weight should be set to be

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

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

$$\beta_{i,j+1} = \beta_{i,j} \frac{1}{1 - q_{i,j+1}} \frac{f(p, \omega, \omega') |\cos \theta'|}{p(\omega')}.$$
 (16.16)

AbsDot() 64
 AtomicFloat 1086
`AtomicFloat::Add()` 1087
 BSDF::f() 575
`CoefficientSpectrum::nSamples` 318
`Interaction::SpawnRay()` 232
 Ray::d 73
`RayDifferential` 75
 Spectrum 315
`SPPMPixel::Phi` 987
`SPPMPixel::vp` 979
 Vector3f 60
`VisiblePoint::bsdf` 979
`VisiblePoint::wo` 979

```
{Sample new photon ray direction} ≡ 986
    {Compute BSDF at photon intersection point 988}
    {Sample BSDF fr and direction wi for reflected photon 988}
    Spectrum bnew = beta * fr * AbsDot(wi, iesect.shading.n) / pdf;
    {Possibly terminate photon path with Russian roulette 989}
    photonRay = (RayDifferential)iesect.SpawnRay(wi);
```

As before, a `nullptr`-valued BSDF * indicates an intersection that should be ignored.

```
(Compute BSDF at photon intersection point) ≡ 987
isect.ComputeScatteringFunctions(photonRay, arena, true,
                                  TransportMode::Importance);

if (!isect.bsdf) {
    --depth;
    photonRay = isept.SpawnRay(photonRay.d);
    continue;
}
const BSDF &photonBSDF = *isect.bsdf;
```

Sampling the BSDF to find the scattered photon direction follows the usual model.

```
(Sample BSDF fr and direction wi for reflected photon) ≡ 987
Vector3f wi, wo = -photonRay.d;
Float pdf;
BxDFType flags;
(Generate bsdfSample for outgoing photon sample 988)
Spectrum fr = photonBSDF.Sample_f(wo, &wi, bsdfSample,
                                   &pdf, BSDF_ALL, &flags);
if (fr.IsBlack() || pdf == 0.f) break;
```

The next two dimensions of the Halton sample vector are used for the BSDF sample.

```
(Generate bsdfSample for outgoing photon sample) ≡ 988
Point2f bsdfSample(RadicalInverse(haltonDim,      haltonIndex),
                     RadicalInverse(haltonDim + 1, haltonIndex));
haltonDim += 2;
```

The photon scattering step should 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 $\beta_{i,j+1}$ is computed using Equation (16.16). Then the ratio of the luminance of $\beta_{i,j+1}$ to the luminance of

BSDF 572

BSDF::Sample_f() 832

BSDF_ALL 513

BxDFType 513

Float 1062

Interaction::SpawnRay() 232

Point2f 68

RadicalInverse() 444

Ray::d 73

Spectrum 315

Spectrum::IsBlack() 317

SurfaceInteraction::bsdf 250

SurfaceInteraction::ComputeScatteringFunctions()
578

TransportMode::Importance
960

Vector3f 60

the photon's old weight $\beta_{i,j}$ is used to set the probability of continuing the path after applying Russian roulette.

The termination probability q 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 $\beta_{i,j+1} > \beta_{i,j}$, as can happen when the ratio of the BSDF's value and the PDF is greater than 1.)

```
{Possibly terminate photon path with Russian roulette} ≡ 987
    Float q = std::max((Float)0, 1 - bnew.y() / beta.y());
    if (RadicalInverse(haltonDim++, haltonIndex) < q)
        break;
    beta = bnew / (1 - q);
```

After all of the photons for the iteration have been traced, the estimate of the incident radiance visible in each pixel area can now be updated based on contributions from photons in the current pass.

```
{Update pixel values from this pass's photons} ≡ 975
    for (int i = 0; i < nPixels; ++i) {
        SPPMPixel &p = pixels[i];
        if (p.M > 0) {
            (Update pixel photon count, search radius, and τ from photons 989)
            p.M = 0;
            for (int j = 0; j < Spectrum::nSamples; ++j)
                p.Phi[j] = (Float)0;
        }
        (Reset VisiblePoint in pixel 990)
    }
```

```
CoefficientSpectrum::nSamples
    318
Float 1062
RadicalInverse() 444
Spectrum 315
Spectrum::y() 325
SPPMPixel 974
SPPMPixel::M 987
SPPMPixel::N 990
SPPMPixel::Phi 987
SPPMPixel::radius 974
SPPMPixel::tau 990
SPPMPixel::vp 979
VisiblePoint::beta 979
```

Equation (16.13) gives the rules to update the search radius and other quantities related to the photon estimate.

```
{Update pixel photon count, search radius, and τ from photons} ≡ 989
    Float gamma = (Float)2 / (Float)3;
    Float Nnew = p.N + gamma * p.M;
    Float Rnew = p.radius * std::sqrt(Nnew / (p.N + p.M));
    Spectrum Phi;
    for (int j = 0; j < Spectrum::nSamples; ++j)
        Phi[j] = p.Phi[j];
    p.tau = (p.tau + p.vp.beta * Phi) *
        (Rnew * Rnew) / (p.radius * p.radius);
    p.N = Nnew;
    p.radius = Rnew;
```

Note that the number of photons that have contributed to the pixel N is actually stored as a `Float`. This quantity must be treated as continuously valued, not a discrete integer, for the progressive radiance estimate to converge to the correct value in the limit.

(SPPMPixel Public Data) \equiv

974

```
Float N = 0;
Spectrum tau;
```

Before the next SPPM iteration begins, it's necessary to zero out the visible point in the pixel so that we don't attempt to re-use this one if no visible point and BSDF * is found in the next iteration.

(Reset VisiblePoint in pixel) \equiv

989

```
p.vp.beta = 0.;
p.vp.bsdf = nullptr;
```

Most of the *(Periodically store SPPM image in film and write image)* fragment is a straightforward matter of allocating an image of `Spectrum` values to pass to `Film::SetImage()` and then initializing the pixels in the image and before calling `Film::WriteImage()`. We won't include that boilerplate here, in order to focus on the last step of the SPPM algorithm, which combines the direct and indirect radiance estimates.

As described earlier, the direct lighting estimate needs to be divided by the number of pixel samples (which in turn is how many iterations have completed at this point) to get its average value. The indirect photon term is computed using Equation (16.11)—the two values then just need to be added together.

(Compute radiance L for SPPM pixel pixel) \equiv

```
const SPPMPixel &pixel = pixels[(y - pixelBounds.pMin.y) * (x1 - x0) +
(x - x0)];
Spectrum L = pixel.Ld / (iter + 1);
L += pixel.tau / (Np * Pi * pixel.radius * pixel.radius);
```

16.3 BI DIRECTIONAL PATH TRACING

The path-tracing algorithm described in Section 14.5 was the first fully general light transport algorithm in computer graphics, handling both a wide variety of geometric representations, lights, and BSDF models. Although it works well for many scenes, path tracing can exhibit high variance in the presence of particular tricky lighting conditions. For example, consider the setting shown in Figure 16.12: 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. The resulting *bidirectional path-tracing*

`Film::SetImage()` 494

`Float` 1062

`Pi` 1063

`Spectrum` 315

`SPPMPixel` 974

`SPPMPixel::ld` 978

`SPPMPixel::radius` 974

`SPPMPixel::tau` 990

`SPPMPixel::vp` 979

`VisiblePoint::beta` 979

`VisiblePoint::bsdf` 979

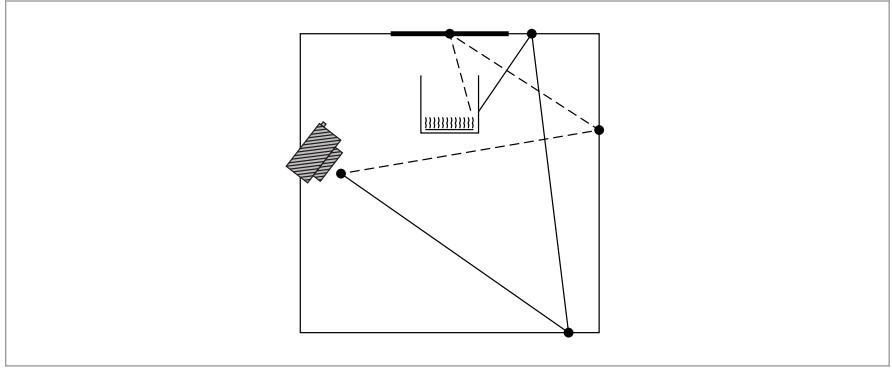


Figure 16.12: A Difficult Case for Path Tracing Starting from the Camera. A light source is illuminating a small area on the ceiling (thick line) such that only paths with a second-to-last vertex in the area indicated will be able to find illumination from the light. Bidirectional methods, where a path is started from the light and is connected with a path from the camera, can handle situations like these more robustly.

algorithm (henceforth referred to as BDPT) is a generalization of the standard path-tracing algorithm that can be much more efficient. In contrast to stochastic progressive photon mapping, BDPT is unbiased and does not blur the scene illumination.

BDPT first incrementally constructs a *camera subpath* starting with a point on the camera p_0 . The next vertex, p_1 , is found by computing the first intersection along the camera ray. Another vertex is found by sampling the BSDF at p_1 and tracing a ray to find a point p_2 , and so forth. The resulting path of t vertices is p_0, p_1, \dots, p_{t-1} . Following the same process starting from a point on a light source q_0 (and then using adjoint BSDFs at each vertex) creates a *light subpath* of s vertices, q_0, q_1, \dots, q_{s-1} .

Given the two subpaths, a complete light-carrying path can be found by connecting a pair of vertices from each path.

$$\bar{p} = q_0, \dots, q_{s'-1}, p_{t'-1}, \dots, p_0,$$

where $s' \leq s$ and $t' \leq t$. (Our notation orders the vertices in \bar{p} according to the propagation of light). If a visibility ray between $q_{s'}$ and $p_{t'}$ is unoccluded, then the path contribution can be found by evaluating the BSDFs at the connecting vertices (see Figure 16.13). More generally, these subpaths can be combined using the theory of path-space integration from Section 14.4.4.

Superficially, this approach bears some semblance to the two phases of the photon mapper; a key difference is that BDPT computes an unbiased estimate without density estimation. There are also significant differences in how the fact that a given light path could have been constructed in multiple different ways is handled.

There are three refinements to the algorithm that we've described so far 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.

- First, subpaths can be reused: given a path $q_0, \dots, q_{s-1}, p_{t-1}, \dots, p_0$, transport can be evaluated over all of the paths given by connecting all the various

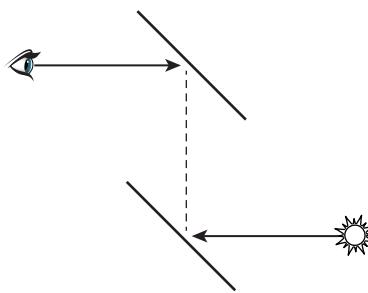


Figure 16.13: Bidirectional path tracing is based on generating two subpaths, one starting from a light and the other starting from the camera. Light-carrying paths can be found by attempting to connect pairs of vertices, one from each path. If a visibility ray between the two (dashed line) is unoccluded, the path's contribution can be added to the radiance estimate.

combinations of prefixes of the two paths together. If two paths have s and t vertices, respectively, then a variety of unique paths can be constructed from them, ranging in length from 2 to $s + t$ vertices long. Figure 16.14 illustrates these strategies for the case of direct illumination.

- The second optimization is not to try to connect paths that use only a single vertex from one of the subpaths. It is preferable to generate those paths using optimized sampling routines provided by the camera and light sources; for light sources, these are the direct lighting techniques that were introduced in Section 14.3.
- The third optimization weights the various strategies for generating paths of a given length more carefully than just averaging all of the strategies that construct paths of the same length. BDPT’s approach of connecting subpaths means that a path containing n scattering events can be generated in $n + 3$ different ways. We can expect that some strategies will be a good choice for producing certain types of paths while being quite poor for others. Multiple importance sampling can be applied to combine the set of connection strategies into a single estimator that uses each strategy where it is best. This application of MIS is crucial to BDPT’s efficiency.

One of BDPT’s connection strategies is to directly connect light subpath vertices to the camera: these paths almost always create a contribution whose raster coordinates differ from the current pixel being rendered, which violates the expectations of the SamplerIntegrator interface. Thus, BDPTIntegrator derives from the more general Integrator interface so that it can have more flexibility in how it updates the image. Its implementation is in the files integrators/bdpt.h and integrators/bdpt.cpp.

```
(BDPT Declarations) ≡
class BDPTIntegrator : public Integrator {
public:
    (BDPTIntegrator Public Methods 994)
private:
    (BDPTIntegrator Private Data 993)
};
```

BDPTIntegrator 992
Integrator 25
SamplerIntegrator 25

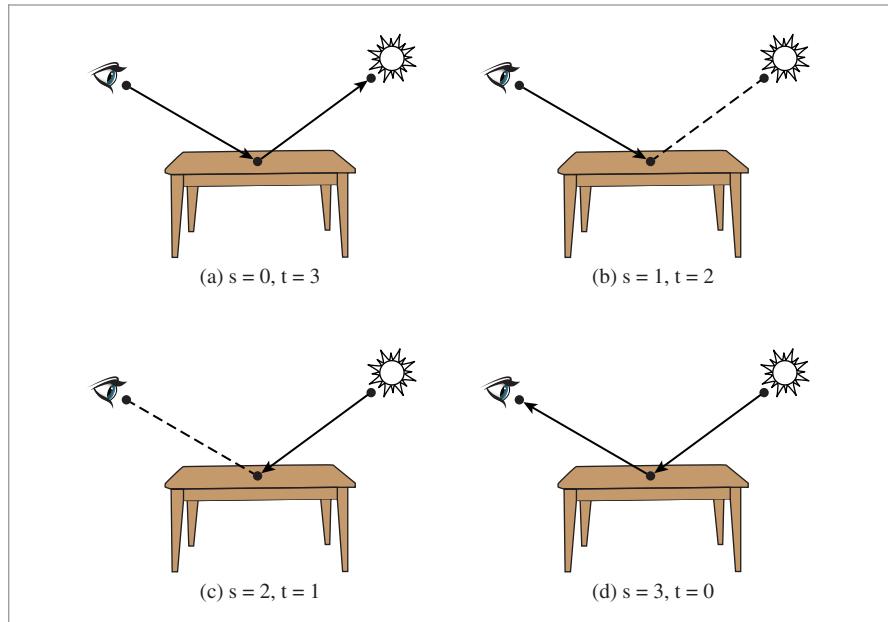


Figure 16.14: The four different ways in which bidirectional path tracing can create a direct illumination path. (a) Standard path tracing without direct illumination sampling, where a ray generated using BSDF sampling from a point on a surface happens to intersect a light source. (b) Path tracing with direct illumination sampling, where the explicit shadow ray is indicated with a dashed line. (c) Particle tracing from a light source with an explicit visibility test between a point on a surface and the camera. (d) Particle tracing where a particle happens to intersect the camera’s lens.

The BDPTIntegrator constructor, which is straightforward and not included here, initializes member variables with the provided camera, sampler, and the maximum path depth.

```
(BDPTIntegrator Private Data) ≡
    std::shared_ptr<Sampler> sampler;
    std::shared_ptr<const Camera> camera;
    const int maxDepth;
```

992

BDPTIntegrator::Render() 994
 BDPTIntegrator::sampler 993
 Camera 356
 MemoryArena 1074
 Sampler 421
 Sampler::Clone() 424
 Sampler::StartNextSample()
 424
 SamplerIntegrator::Render()
 26

All subpath creation and connection steps are performed in a nested parallel loop over pixels in BDPTIntegrator::Render(). The overall structure of this method is very similar to SamplerIntegrator::Render():

- The image is subdivided into tiles of 16×16 pixels, which are processed in parallel.
- For each tile, the method declares a MemoryArena, arena and acquires a Sampler instance from BDPTIntegrator::sampler via a call to Sampler::Clone().
- It then loops over the pixels in each tile, taking samples from each one until Sampler::StartNextSample() returns false, at which point it advances to the next pixel.

We won't include this code here, as the details should be familiar now. Instead, we'll move forward to the fragment responsible for generating and connecting the subpaths for a pixel sample.

(BDPTIntegrator Public Methods) ≡
 void Render(const Scene &scene);

992

Generating a single BDPT sample involves sampling a pixel position in the image, generating camera and light subpaths, and then connecting them using a variety of specialized connection strategies.

(Generate a single sample using BDPT) ≡
 Point2f pFilm = (Point2f)pPixel + tileSampler->Get2D();
(Trace the camera and light subpaths 994)
(Execute all BDPT connection strategies 995)

The `Vertex` class, which will be introduced in Section 16.3.1, represents a vertex along a subpath. We start by allocating two arrays for vertices for the two subpaths. In addition to a vertex on a surface, a `Vertex` can represent a vertex for a scattering event in participating media, a vertex on a light source, or a vertex on the camera lens.

For each subpath, one more vertex than the maximum path length must be allocated to store the starting vertex on the light or camera. Camera subpaths get yet again one more vertex, which allows camera paths to randomly intersect light sources—this strategy is important for rendering area lights seen by reflections only from specular surfaces, for example. (The corresponding strategy of allowing a light subpath to randomly intersect the camera lens is less useful in practice.)

The `GenerateCameraSubpath()` and `GenerateLightSubpath()` functions, which generate these two subpaths, will be defined in Section 16.3.2, after some preliminaries related to the `Vertex` representation.

(Trace the camera and light subpaths) ≡
 Vertex *cameraVertices = arena.Alloc<Vertex>(maxDepth + 2);
 Vertex *lightVertices = arena.Alloc<Vertex>(maxDepth + 1);
 int nCamera = GenerateCameraSubpath(scene, *tileSampler, arena,
 maxDepth + 2, *camera, pFilm, cameraVertices);
 int nLight = GenerateLightSubpath(scene, *tileSampler, arena,
 maxDepth + 1, cameraVertices[0].time(), *lightDistr, lightVertices);

994

After the subpaths have been generated, a nested `for` loop iterates over all pairs of vertices from the two subpaths and attempts to connect them. In these loops, s and t correspond to the number of vertices to use from the corresponding subpath; an index of 0 means that no scattering events are used from the corresponding subpath. In our implementation, this strategy is only supported for the $s = 0$ case, which furthermore requires `cameraVertices[t]` to be a surface intersection involving a light source. Because the dual case—intersecting a camera with $t = 0$ —is not supported, the loop over camera subpaths starts at $t = 1$.

A path length of 1 corresponds to connecting a point on the camera lens or a light source to the other subpath. For light endpoints, this is identical to the standard light

BDPTIntegrator::maxDepth 993
`GenerateCameraSubpath()` 1003
`GenerateLightSubpath()` 1004
`MemoryArena::Alloc()` 1074
`Point2f` 68
`Sampler::Get2D()` 422
`Scene` 23
`Vertex` 996
`Vertex::time()` 997

sampling approach provided by `Light::Sample_Li()` and first used in Section 14.3.1; our implementation uses this existing functionality. For camera endpoints, we will rely on the symmetric analog `Camera::Sample_Wi()`. Since `Camera::Sample_Wi()` and `Light::Sample_Li()` cannot both be used at the same time, we skip the $s = t = 1$ case.

(Execute all BDPT connection strategies) ≡ 994

```
Spectrum L(0.f);
for (int t = 1; t <= nCamera; ++t) {
    for (int s = 0; s <= nLight; ++s) {
        int depth = t + s - 2;
        if ((s == 1 && t == 1) || depth < 0 || depth > maxDepth)
            continue;
        (Execute the (s, t) connection strategy and update L 995)
    }
}
filmTile->AddSample(pFilm, L);
```

The `ConnectBDPT()` function attempts to connect the two subpaths with the given number of vertices; it returns the weighted contribution of the radiance carried along the resulting path. (It will be defined shortly, in Section 16.3.3.) In most cases, this contribution is accumulated into the variable `L` that will be provided to the `FilmTile` after all of the subpath connections have been attempted. However, the $t = 1$ connection connects a vertex of the light subpath directly to the camera and thus will produce different raster positions in every iteration—in this case, the implementation calls `Film::AddSplat()` to immediately record its sample contribution.

(Execute the (s, t) connection strategy and update L) ≡ 995

```
Point2f pFilmNew = pFilm;
Float misWeight = 0.f;
Spectrum Lpath = ConnectBDPT(scene, lightVertices, cameraVertices, s, t,
    *lightDistr, *camera, *tileSampler, &pFilmNew, &misWeight);
if (t != 1)
    L += Lpath;
else
    film->AddSplat(pFilmNew, Lpath);
```

16.3.1 VERTEX ABSTRACTION LAYER

`BDPTIntegrator::maxDepth` 993
`Camera::Sample_Wi()` 954
`ConnectBDPT()` 1009
`Film::AddSplat()` 494
`FilmTile` 489
`FilmTile::AddSample()` 490
`Float` 1062
`Light::Sample_Li()` 716
`Point2f` 68
`Spectrum` 315

A general advantage of path-space rendering techniques is their ability to create paths in a large number of different ways, but this characteristic often leads to cluttered and hard-to-debug implementations. Establishing connections between pairs of vertices on the light and camera subpaths is a simple operation when only surface interactions are involved but quickly becomes unwieldy if one or both of the vertices may represent a scattering event in participating media, for example.

Instead of an inconveniently large number of conditional statements in the core BDPT code, we'll define the `Vertex` type, which can represent any kind of path vertex. All of the necessary conditional logic to handle various cases that occur throughout the BDPT implementation will be encapsulated in its methods.

```
(BDPT Declarations) +≡
  struct Vertex {
    (Vertex Public Data 996)
    (Vertex Public Methods 997)
  };
(Vertex Public Data) ≡
  VertexType type;
```

996

Altogether, four different types of path vertices are supported in pbrt.

```
(BDPT Helper Definitions) ≡
  enum class VertexType { Camera, Light, Surface, Medium };
```

The beta member variable is analogous to the β variable in the volumetric path tracer (Section 15.3.1) or the weight carried by particles in the SPPMIntegrator: it contains the product of the BSDF or phase function values, transmittances, and cosine terms for the vertices in the path generated so far, divided by their respective sampling PDFs. For the light subpath, they also include the emitted radiance divided by the density of the emission position and direction. For the camera subpath, radiance is replaced by importance.

```
(Vertex Public Data) +≡
  Spectrum beta;
```

996

Instances of various types of Interactions represent type-specific data about the vertex. This information is arranged as a space-efficient C++ union since only one of the entries is used at a time.

```
(Vertex Public Data) +≡
  union
  {
    EndpointInteraction ei;
    MediumInteraction mi;
    SurfaceInteraction si;
  };

```

996

EndpointInteraction is a new interaction implementation that is used only by BDPT. It records the position of a path endpoint—i.e., a position on a light source or the lens of the camera—and stores a pointer to the camera or light in question.

```
(EndpointInteraction Declarations) ≡
  struct EndpointInteraction : Interaction {
    union {
      const Camera *camera;
      const Light *light;
    };
    (EndpointInteraction Public Methods 997)
  };

```

Camera 356
 EndpointInteraction 996
 Interaction 115
 Light 714
 MediumInteraction 688
 Spectrum 315
 SPPMIntegrator 973
 SurfaceInteraction 116
 VertexType 996

There are a multiple constructors that initialize the EndpointInteraction contents using a pointer and either an existing Interaction or a sampled ray. For brevity, we only show the constructors for light endpoints.

(EndpointInteraction Public Methods) ≡ 996

```
EndpointInteraction(const Light *light, const Ray &r, const Normal3f &n)
: Interaction(r.o, r.time, r.medium), light(light) { n = n; }
```

(EndpointInteraction Public Methods) +≡ 996

```
EndpointInteraction(const Interaction &it, const Light *light)
: Interaction(it), light(light) { }
```

A range of static helper functions create `Vertex` instances for the various types of path vertices. We'll only include their declarations here, as their implementations are all straightforward. We could instead have provided a range of overloaded constructors that took various `Interaction` types as parameters, but we think that having the name of the type of vertex being created explicit in a function call makes the following code easier to read.

(Vertex Public Methods) ≡ 996

```
static inline Vertex CreateCamera(const Camera *camera, const Ray &ray,
const Spectrum &beta);
static inline Vertex CreateCamera(const Camera *camera,
const Interaction &it, const Spectrum &beta);
static inline Vertex CreateLight(const Light *light, const Ray &ray,
const Normal3f &nLight, const Spectrum &Le, Float pdf);
static inline Vertex CreateLight(const EndpointInteraction &ei,
const Spectrum &beta, Float pdf);
static inline Vertex CreateMedium(const MediumInteraction &mi,
const Spectrum &beta, Float pdf, const Vertex &prev);
static inline Vertex CreateSurface(const SurfaceInteraction &si,
const Spectrum &beta, Float pdf, const Vertex &prev);
```

Camera 356
 EndpointInteraction 996
 Float 1062
 Interaction 115
 Interaction::n 116
 Interaction::p 115
 Light 714
 Medium 684
 MediumInteraction 688
 Normal3f 71
 Point3f 68
 Ray 73
 Ray::medium 74
 Ray::time 73
 Spectrum 315
 SurfaceInteraction 116
 Vertex 996
 Vertex::ei 996
 Vertex::GetInteraction() 997
 Vertex::mi 996
 Vertex::si 996
 Vertex::type 996
 VertexType::Medium 996
 VertexType::Surface 996

It is often necessary to access the core fields in `Interaction` that are common to all types of vertices; the `Vertex::GetInteraction()` method extracts this shared part. Since `Vertex::mi`, `Vertex::si`, and `Vertex::ei` all derive from `Interaction` and are part of the same union and thus their base `Interactions` are at the same location in memory, the conditional logic below should be removed by the compiler.

(Vertex Public Methods) +≡ 996

```
const Interaction &GetInteraction() const {
    switch (type) {
        case VertexType::Medium: return mi;
        case VertexType::Surface: return si;
        default: return ei;
    }
}
```

The convenience function `Vertex::p()` returns the vertex position. We omit definitions of `Vertex::time()`, `Vertex::ng()`, and `Vertex::ns()`, which are defined analogously, and return the time, geometric normal, and shading normal, respectively, of the vertex.

(Vertex Public Methods) +≡ 996

```
const Point3f &p() const { return GetInteraction().p; }
```

The `delta` attribute is only used by surface interactions and records whether a Dirac delta function was sampled (e.g., when light is scattered by a perfectly specular material).

```
(Vertex Public Data) +≡  
    bool delta = false;
```

A simple way to find out whether a vertex (including endpoints) is located on a surface is to check whether `Vertex::ng()` returns a nonzero result.

```
(Vertex Public Methods) +≡  
    bool IsOnSurface() const { return ng() != Normal3f(); }
```

`Vertex::f()` evaluates the portion of the measurement equation, (16.1), associated with a vertex. This method only needs to handle surface and medium vertices since the BDPT implementation only invokes it in those cases. Note that the next vertex in the path is the only one passed to this method: though the direction to the predecessor vertex is needed to evaluate the BRDF or phase function, this information is already available in `Interaction::wo` from when the `Vertex` was first created.

```
(Vertex Public Methods) +≡  
    Spectrum f(const Vertex &next) const {  
        Vector3f wi = Normalize(next.p() - p());  
        switch (type) {  
            case VertexType::Surface: return si.bsdf->f(si.wo, wi);  
            case VertexType::Medium: return mi.phase->p(mi.wo, wi);  
        }  
    }
```

The `Vertex::IsConnectible()` method returns a Boolean value that indicates whether a connection strategy involving the current vertex can in principle succeed. If, for example, the vertex is a surface interaction whose BSDF only consists of Dirac delta components, then we can never successfully connect it to a subpath vertex in the other path: there's zero probability of choosing a direction where the delta distribution is nonzero. The implementation assumes that medium and camera vertices are always connectible (the latter assumption would have to be modified if support for orthographic cameras is added).

```
(Vertex Public Methods) +≡  
    bool IsConnectible() const {  
        switch (type) {  
            case VertexType::Medium: return true;  
            case VertexType::Light: return  
                (ei.light->flags & (int)LightFlags::DeltaDirection) == 0;  
            case VertexType::Camera: return true;  
            case VertexType::Surface: return si.bsdf->NumComponents(  
                BxDFType(BSDF_DIFFUSE | BSDF_GLOSSY |  
                          BSDF_REFLECTION | BSDF_TRANSMISSION)) > 0;  
        }  
    }
```

BSDF::f() 575
BSDF::NumComponents() 573
BSDF_DIFFUSE 513
BSDF_GLOSSY 513
BSDF_REFLECTION 513
BSDF_TRANSMISSION 513
BxDFType 513
Camera 356
EndpointInteraction::light 996
Interaction::wo 115
Light 714
Light::flags 715
LightFlags::DeltaDirection 715
Medium 684
MediumInteraction::phase 688
Normal3f 71
PhaseFunction::p() 681
Spectrum 315
SurfaceInteraction::bsdf 250
Vector3::Normalize() 66
Vector3f 60
Vertex 996
Vertex::f() 998
Vertex::IsConnectible() 998
Vertex::mi 996
Vertex::ng() 997
Vertex::p() 997
Vertex::si 996
Vertex::type 996
VertexType::Camera 996
VertexType::Light 996
VertexType::Medium 996
VertexType::Surface 996

A few helper methods are useful for working with lights—these are necessary to deal with the considerable variety of light sources supported by pbrt.

For instance, when the Primitive underlying a surface interaction vertex is itself an area light, the vertex can take on different roles depending on the BDPT connection strategy: it can be re-interpreted as a light source and used as a path endpoint, or it can serve as a normal scattering event to generate paths of greater length. The `Vertex::IsLight()` method therefore provides a comprehensive test for whether a vertex can be interpreted as a light source.

```
(Vertex Public Methods) +≡
    bool IsLight() const {
        return type == VertexType::Light ||
            (type == VertexType::Surface && si.primitive->GetAreaLight());
    }
```

996

Light sources that have an emission profile that contains a Dirac delta distribution must be treated specially in the computation of multiple importance sampling weights; `Vertex::IsDeltaLight()` checks for this case.

```
(Vertex Public Methods) +≡
    bool IsDeltaLight() const {
        return type == VertexType::Light && ei.light &&
            ::IsDeltaLight(ei.light->flags);
    }
```

996

`AreaLight` 734
`AreaLight::L()` 734
`EndpointInteraction::light` 996
`InfiniteAreaLight` 737
`IsDeltaLight()` 715
`Light` 714
`Light::flags` 715
`LightFlags::Infinite` 715
`Primitive` 248
`Primitive::GetAreaLight()` 249
`Scene` 23
`Spectrum` 315
`SurfaceInteraction::primitive` 249
`Vector3::Normalize()` 66
`Vector3f` 60
`Vertex` 996
`Vertex::IsDeltaLight()` 999
`Vertex::IsInfiniteLight()` 999
`Vertex::IsLight()` 999
`Vertex::p()` 997
`Vertex::si` 996
`Vertex::type` 996
`VertexType::Light` 996
`VertexType::Surface` 996

The `Vertex::IsInfiniteLight()` method indicates whether a vertex is associated with an infinite area light. Such vertices can be created by sampling an emitted ray from an `InfiniteAreaLight` or by tracing a ray from the camera that escapes into the environment. In the latter case, the vertex is marked with the type `VertexType::Light`, but `ei.light` stores `nullptr` since no specific light source was intersected.

```
(Vertex Public Methods) +≡
    bool IsInfiniteLight() const {
        return type == VertexType::Light &&
            (!ei.light || ei.light->flags & (int)LightFlags::Infinite);
    }
```

996

Finally, `Le()` can be used to find emitted radiance from an intersected light source toward another vertex.

```
(Vertex Public Methods) +≡
    Spectrum Le(const Scene &scene, const Vertex &v) const {
        if (!IsLight()) return Spectrum(0.f);
        Vector3f w = Normalize(v.p() - p());
        if (IsInfiniteLight()) {
            (Return emitted radiance for infinite light sources 1000)
        } else {
            const AreaLight *light = si.primitive->GetAreaLight();
            return light->L(si, w);
        }
    }
```

996

(Return emitted radiance for infinite light sources) \equiv 999

```
Spectrum Le(0.f);
for (const auto &light : scene.lights)
    Le += light->Le(Ray(p(), -w));
return Le;
```

Probability Densities

BDPT's multiple importance sampling code requires detailed information about the probability density of light-carrying paths with respect to a range of different path sampling strategies. It is crucial that these densities are expressed in the same probability measure so that ratios of their probabilities are meaningful. The implementation here uses the *area product measure* for path probabilities. It expresses the density of a path as the product of the densities of its individual vertices, which are in turn given in a simple common (and consistent) measure: *probability per unit area*.⁹ This is the same measure as was initially used to derive the surface form of the LTE in Section 14.4.3.

Recall from Section 5.5 that the Jacobian of the mapping from solid angles to surface area involves the inverse squared distance and the cosine of angle between the geometric normal at next and w (assuming next is a surface vertex—if it is a point in a participating medium, there is no cosine term (Section 15.1.1)). The ConvertDensity() method returns the product of this Jacobian (computed from the vertex attributes) and the pdf parameter, which should express a solid angle density at the vertex. (Infinite area light sources need special handling here; this case is discussed later, in Section 16.3.5.)

(Vertex Public Methods) $+ \equiv$ 996

```
Float ConvertDensity(Float pdf, const Vertex &next) const {
    (Return solid angle density if next is an infinite area light 1020)
    Vector3f w = next.p() - p();
    Float invDist2 = 1 / w.LengthSquared();
    if (next.IsOnSurface())
        pdf *= AbsDot(next.ng(), w * std::sqrt(invDist2));
    return pdf * invDist2;
}
```

Each vertex has two densities: the first, pdfFwd, stores *forward* density of the current vertex, which is the probability per unit area of the current vertex as generated by the path sampling algorithm. The second density, pdfRev, is the hypothetical probability density of the vertex if the direction of light transport was *reversed*—that is, if radiance transport was used in place of importance transport for the camera path and vice versa for the light path. This reverse density will be crucial for computing MIS weights in Section 16.3.4.

(Vertex Public Data) $+ \equiv$ 996

```
Float pdfFwd = 0, pdfRev = 0;
```

[AbsDot\(\)](#) 64
[Float](#) 1062
[Light::Le\(\)](#) 741
[Ray](#) 73
[Scene::lights](#) 23
[Spectrum](#) 315
[Vector3::LengthSquared\(\)](#) 65
[Vector3f](#) 60
[Vertex](#) 996
[Vertex::IsOnSurface\(\)](#) 998
[Vertex::ng\(\)](#) 997
[Vertex::p\(\)](#) 997

⁹ Note that an analogously defined “product solid angle measure” would not satisfy the requirement of a common and consistent measure: solid angle densities are always expressed with respect to a specific vertex position—relating the densities as “seen” from different vertices would require additional Jacobian factors to account for the underlying change of variables.

The `Vertex::Pdf()` method returns the probability per unit area of the sampling technique associated with a given vertex. Given a preceding vertex `prev`, it evaluates the density for sampling the vertex next for rays leaving the vertex `*this`. The `prev` argument may be equal to `nullptr` for path endpoints (i.e., cameras or light sources), which have no predecessor. Light sources require some extra care and are handled separately via the `PdfLight()` method that will be discussed shortly.

```
(Vertex Public Methods) +≡ 996
    Float Pdf(const Scene &scene, const Vertex *prev,
              const Vertex &next) const {
        if (type == VertexType::Light)
            return PdfLight(scene, next);
(Compute directions to preceding and next vertex 1001)
(Compute directional density depending on the vertex type 1001)
(Return probability per unit area at vertex next 1001)
    }
```

For all other vertex types, the function first computes normalized directions to the preceding and next vertex (if present).

```
(Compute directions to preceding and next vertex) ≡ 1001
    Vector3f wp, wn = Normalize(next.p() - p());
    if (prev)
        wp = Normalize(prev->p() - p());
```

Depending on the vertex type, `Pdf()` invokes the appropriate PDF method and stores the probability per unit solid angle for sampling the direction to next in the variable `pdf`.

```
(Compute directional density depending on the vertex type) ≡ 1001
    Float pdf, unused;
    if (type == VertexType::Camera)
        ei.camera->Pdf_We(ei.SpawnRay(wn), &unused, &pdf);
    else if (type == VertexType::Surface)
        pdf = si.bsdf->Pdf(wp, wn);
    else if (type == VertexType::Medium)
        pdf = mi.phase->p(wp, wn);
```

Finally, the solid angle density is converted to a probability per unit area at next.

```
(Return probability per unit area at vertex next) ≡ 1001
    return ConvertDensity(pdf, next);
```

Light-emitting vertices can be created in two different ways: by using a sampling routine like `Light::Sample_Le()`, or by intersecting an emissive surface via ray tracing. To be able to compare these different strategies as part of a multiple importance sampling scheme, it is necessary to know the corresponding probability per unit area for a light vertex. This task is handled by the `PdfLight()` method.

Its definition resembles that of `Vertex::Pdf()`: it computes the direction from the current vertex to the provided vertex and invokes `Light::Pdf_Le()` to retrieve the solid angle density of the underlying sampling strategy, which is subsequently converted into a

density per unit area at v . In contrast to `Vertex::Pdf()`, this method also treats surface vertices located on area lights as if they were light source vertices. Once more, there is a special case for infinite area lights, which we postpone until Section 16.3.5.

```
(Vertex Public Methods) +≡ 996
Float PdfLight(const Scene &scene, const Vertex &v) const {
    Vector3f w = v.p() - p();
    Float invDist2 = 1 / w.LengthSquared();
    w *= std::sqrt(invDist2);
    Float pdf;
    if (IsInfiniteLight()) {
        ⟨Compute planar sampling density for infinite light sources 1022⟩
    } else {
        ⟨Get pointer light to the light source at the vertex 1002⟩
        ⟨Compute sampling density for non-infinite light sources 1002⟩
    }
    if (v.IsOnSurface())
        pdf *= AbsDot(v.ng(), w);
    return pdf;
}
```

Depending on the vertex type, the pointer to the light source implementation must be obtained from one of two different locations.

```
(Get pointer light to the light source at the vertex) ≡ 1002, 1003
const Light *light = type == VertexType::Light ?
    ei.light : si.primitive->GetAreaLight();

(Compute sampling density for non-infinite light sources) ≡ 1002
Float pdfPos, pdfDir;
light->Pdf_Le(Ray(p(), w, time()), ng(), &pdfPos, &pdfDir);
pdf = pdfDir * invDist2;
```

By symmetry, we would now expect a dual routine `Vertex::PdfCamera()` that applies to camera endpoints. However, cameras in `pbrt` are never represented using explicit geometry: thus, they cannot be reached by ray intersections, which eliminates the need for a dedicated query function. If desired, a perfectly symmetric implementation could be achieved by instantiating scene geometry that is tagged with an “area camera” analogous to area lights. This increases the set of possible BDPT connection strategies, though their benefit is negligible in most scenes due to the low probability of intersecting the camera.

Note that the `Pdf()` and `PdfLight()` methods use the directional probability density of the importance strategy implemented at the current vertex as measured at the location of another given vertex. However, this is not enough to fully characterize the behavior of path endpoints, whose sampling routines generate rays from a 4D distribution. An additional `PdfLightOrigin()` method fills the gap by providing information about the spatial distribution of samples on the light sources themselves. For the same reason as before, a dedicated `PdfCameraOrigin()` method for camera endpoints is not needed.

AbsDot() 64
 EndpointInteraction::light 996
 Float 1062
 Light 714
 Light::Pdf_Le() 955
 Primitive::GetAreaLight() 249
 Ray 73
 Scene 23
 SurfaceInteraction::primitive 249
 Vector3::LengthSquared() 65
 Vector3f 60
 Vertex 996
 Vertex::ei 996
 Vertex::IsInfiniteLight() 999
 Vertex::IsOnSurface() 998
 Vertex::ng() 997
 Vertex::p() 997
 Vertex::Pdf() 1001
 Vertex::si 996
 Vertex::time() 997
 Vertex::type 996
 VertexType::Light 996

```

⟨Vertex Public Methods⟩ +≡ 996
    Float PdfLightOrigin(const Scene &scene, const Vertex &v,
                          const Distribution1D &lightDistr) const {
        Vector3f w = Normalize(v.p() - p());
        if (IsInfiniteLight()) {
            ⟨Return solid angle density for infinite light sources 1021⟩
        } else {
            ⟨Return solid angle density for non-infinite light sources 1003⟩
        }
    }

⟨Return solid angle density for non-infinite light sources⟩ ≡ 1003
    Float pdfPos, pdfDir, pdfChoice = 0;
    ⟨Get pointer light to the light source at the vertex 1002⟩
    ⟨Compute the discrete probability of sampling light, pdfChoice 1003⟩
    light->Pdf_Le(Ray(p(), w, time()), ng(), &pdfPos, &pdfDir);
    return pdfPos * pdfChoice;

```

To determine the discrete probability of choosing light among the available light sources, we must find the pointer to the light source and look up the corresponding entry in lightDistr. If there are very many light sources, the linear search here will be inefficient. In that case, this computation could be implemented more efficiently by storing this probability directly in the light source class.

```

⟨Compute the discrete probability of sampling light, pdfChoice⟩ ≡ 1003
    for (size_t i = 0; i < scene.lights.size(); ++i) {
        if (scene.lights[i].get() == light) {
            pdfChoice = lightDistr.DiscretePDF(i);
            break;
        }
    }

Camera 356
Distribution1D 758
Distribution1D::DiscretePDF() 760
Float 1062
GenerateCameraSubpath() 1003
GenerateLightSubpath() 1004
Light::Pdf_Le() 955
MemoryArena 1074
Point2f 68
RandomWalk() 1005
Ray 73
Sampler 421
Scene 23
Scene::lights 23
Vector3::Normalize() 66
Vector3f 60
Vertex 996
Vertex::IsInfiniteLight() 999
Vertex::ng() 997
Vertex::p() 997
Vertex::time() 997

```

16.3.2 GENERATING THE CAMERA AND LIGHT SUBPATHS

A symmetric pair of functions, GenerateCameraSubpath() and GenerateLightSubpath(), generates the two corresponding types of subpaths. Both do some initial work to get the path started and then call out to a second function, RandomWalk(), which takes care of sampling the following vertices and initializing the path array. Both of these functions return the number of vertices in the subpath.

```

⟨BDPT Utility Functions⟩ +≡
    int GenerateCameraSubpath(const Scene &scene, Sampler &sampler,
                             MemoryArena &arena, int maxDepth, const Camera &camera,
                             const Point2f &pFilm, Vertex *path) {
        if (maxDepth == 0)
            return 0;
        ⟨Sample initial ray for camera subpath 1004⟩
        ⟨Generate first vertex on camera subpath and start random walk 1004⟩
    }

```

A camera path starts with a camera ray from `Camera::GenerateRayDifferential()`. As in the `SamplerIntegrator`, the ray's differentials are scaled so that they reflect the actual pixel sampling density.

<i>(Sample initial ray for camera subpath) ≡</i> <pre> CameraSample cameraSample; cameraSample.pFilm = pFilm; cameraSample.time = sampler.Get1D(); cameraSample.pLens = sampler.Get2D(); RayDifferential ray; Spectrum beta = camera.GenerateRayDifferential(cameraSample, &ray); ray.ScaleDifferentials(1 / std::sqrt(sampler.samplesPerPixel)); </pre>	1003
---	------

The vertex at position `path[0]` is initialized with a special endpoint vertex on the camera lens (for cameras with finite apertures) or pinhole. The `RandomWalk()` function then takes care of generating the rest of the vertices. `TransportMode` reflects the quantity that is carried back to the origin of the path—hence `TransportMode::Radiance` is used here. Since the first element of `path` was already used for the endpoint vertex, `RandomWalk()` is invoked such that it writes sampled vertices starting at position `path[1]` with a maximum depth of `maxDepth - 1`. The function returns the total number of sampled vertices.

<i>(Generate first vertex on camera subpath and start random walk) ≡</i> <pre> Float pdfPos, pdfDir; path[0] = Vertex::CreateCamera(&camera, ray, beta); camera.Pdf_We(ray, &pdfPos, &pdfDir); return RandomWalk(scene, ray, sampler, arena, beta, pdfDir, maxDepth - 1, TransportMode::Radiance, path + 1) + 1; </pre>	1003
--	------

The function `GenerateLightSubpath()` works in a similar fashion, with some minor differences corresponding to the fact that the path starts from a light source.

<i>(BDPT Utility Functions) +≡</i> <pre> int GenerateLightSubpath(const Scene &scene, Sampler &sampler, MemoryArena &arena, int maxDepth, Float time, const Distribution1D &lightDistr, Vertex *path) { if (maxDepth == 0) return 0; <i>(Sample initial ray for light subpath 1005)</i> <i>(Generate first vertex on light subpath and start random walk 1005)</i> } </pre>	1005
--	------

As usual in this integrator, a specific light is chosen by sampling from the provided `Distribution1D`. Next, an emitted ray is sampled via the light's implementation of `Light::Sample_Le()`.

```

Camera::
    GenerateRayDifferential() 357
Camera::Pdf_We() 953
CameraSample 357
CameraSample::pFilm 357
CameraSample::pLens 357
CameraSample::time 357
Distribution1D 758
Float 1062
Light::Sample_Le() 955
MemoryArena 1074
RandomWalk() 1005
RayDifferential 75
RayDifferential::
    ScaleDifferentials() 75
Sampler 421
Sampler::Get1D() 422
Sampler::Get2D() 422
Sampler::samplesPerPixel 422
SamplerIntegrator 25
Scene 23
Spectrum 315
TransportMode 960
TransportMode::Radiance 960
Vertex 996
Vertex::CreateCamera() 997

```

```
(Sample initial ray for light subpath) ≡ 1004
    Float lightPdf;
    int lightNum = lightDistr.SampleDiscrete(sampler.Get1D(), &lightPdf);
    const std::shared_ptr<Light> &light = scene.lights[lightNum];
    RayDifferential ray;
    Normal3f nLight;
    Float pdfPos, pdfDir;
    Spectrum Le = light->Sample_Le(sampler.Get2D(), sampler.Get2D(), time,
                                    &ray, &nLight, &pdfPos, &pdfDir);
    if (pdfPos == 0 || pdfDir == 0 || Le.IsBlack())
        return 0;
```

The beta variable is initialized with the associated sampling weight, which is given by the emitted radiance multiplied by a cosine factor from the light transport equation and divided by the probability of the sample in ray-space. This step is analogous to Equation (16.15) and the approach implemented in the fragment *(Generate photonRay from light source and initialize beta)* from the particle tracing step of the SPPM integrator.

```
(Generate first vertex on light subpath and start random walk) ≡ 1004
    path[0] = Vertex::CreateLight(light.get(), ray, nLight, Le,
                                pdfPos * lightPdf);
    Spectrum beta = Le * AbsDot(nLight, ray.d) / (lightPdf * pdfPos * pdfDir);
    int nVertices = RandomWalk(scene, ray, sampler, arena, beta, pdfDir,
                               maxDepth - 1, TransportMode::Importance,
                               path + 1);
(Correct subpath sampling densities for infinite area lights 1021)
    return nVertices + 1;

AbsDot() 64
Distribution1D::
    SampleDiscrete() 760
Float 1062
Light 714
Light::Sample_Le() 955
MemoryArena 1074
Normal3f 71
RandomWalk() 1005
RayDifferential 75
Sampler 421
Sampler::Get1D() 422
Sampler::Get2D() 422
Scene 23
Scene::lights 23
Spectrum 315
Spectrum::IsBlack() 317
TransportMode 960
TransportMode::Importance 960
Vertex 996
Vertex::CreateLight() 997
```

RandomWalk() traces paths starting at an initial vertex. It assumes that a position and an outgoing direction at the corresponding path endpoint were previously sampled and that this information is provided via the input arguments ray, a path throughput weight beta, and a parameter pdfFwd that gives the probability of sampling the ray per unit solid angle of ray.d. The parameter mode selects between importance and radiance transport (Section 16.1). The path vertices are stored in the provided path array up to a maximum number of maxDepth vertices, and the actual number of generated vertices is returned at the end.

(BDPT Utility Functions) +≡

```
int RandomWalk(const Scene &scene, RayDifferential ray, Sampler &sampler,
               MemoryArena &arena, Spectrum beta, Float pdf, int maxDepth,
               TransportMode mode, Vertex *path) {
    if (maxDepth == 0)
        return 0;
```

```

int bounces = 0;
(Declare variables for forward and reverse probability densities 1006)
while (true) {
    (Attempt to create the next subpath vertex in path 1006)
}
return bounces;
}

```

The two variables `pdfFwd` and `pdfRev` are updated during every loop iteration and satisfy the following invariants: at the beginning of each iteration, `pdfFwd` records the probability per unit solid angle of the sampled ray direction `ray.d`. On the other hand, `pdfRev` denotes the *reverse* probability at the end of each iteration—that is, the density of the opposite light transport mode per unit solid angle along the same ray segment.

(Declare variables for forward and reverse probability densities) ≡ 1005

```

Float pdfFwd = pdf, pdfRev = 0;

```

(Attempt to create the next subpath vertex in path) ≡ 1005

```

MediumInteraction mi;
(Trace a ray and sample the medium, if any 1006)
if (mi.IsValid()) {
    (Record medium interaction in path and compute forward density 1007)
    (Sample direction and compute reverse density at preceding vertex 1007)
} else {
    (Handle surface interaction for path generation 1007)
}
(Compute reverse area density at preceding vertex 1008)

```

The loop body begins by intersecting the current ray against the scene geometry. If the ray is passing through a participating medium, the call to `Medium::Sample()` possibly samples a scattering event between the ray and the surface. It returns the medium sampling weight, which is incorporated into the path contribution weight `beta`.

(Trace a ray and sample the medium, if any) ≡ 1006

```

SurfaceInteraction isect;
bool foundIntersection = scene.Intersect(ray, &isect);
if (ray.medium)
    beta *= ray.medium->Sample(ray, sampler, arena, &mi);
if (beta.IsBlack())
    break;
Vertex &vertex = path[bounces], &prev = path[bounces - 1];

```

Float [1062](#)
`Medium::Sample()` [891](#)
`MediumInteraction` [688](#)
`MediumInteraction::IsValid()` [893](#)
`Ray::medium` [74](#)
`Scene::Intersect()` [24](#)
`Spectrum::IsBlack()` [317](#)
`SurfaceInteraction` [116](#)
`Vertex` [996](#)
`Vertex::CreateMedium()` [997](#)
`Vertex::pdfFwd` [1000](#)

When `Medium::Sample()` generates a medium scattering event, the corresponding `Interaction` is stored in a `Vertex` and appended at the end of the path array. The `Vertex::CreateMedium()` method converts the solid angle density in `pdfFwd` to a probability per unit area and stores the result in `Vertex::pdfFwd`.

```
<Record medium interaction in path and compute forward density> ≡ 1006
    vertex = Vertex::CreateMedium(mi, beta, pdfFwd, prev);
    if (++bounces >= maxDepth)
        break;
```

If the maximum path depth has not yet been exceeded, a scattered direction is sampled from the phase function and used to spawn a new ray that will be processed by the next loop iteration.

At this point, we could evaluate the phase function with swapped arguments to obtain the sampling density at the preceding vertex for a hypothetical random walk that would have produced the same scattering interactions in reverse order. Since phase functions are generally symmetric with respect to their arguments, instead we simply reuse the value computed for pdfFwd.

```
<Sample direction and compute reverse density at preceding vertex> ≡ 1006
    Vector3f wi;
    pdfFwd = pdfRev = mi.phase->Sample_p(-ray.d, &wi, sampler.Get2D());
    ray = mi.SpawnRay(wi);
```

For surfaces, the overall structure is similar, though some extra care is required to deal with non-symmetric scattering and surfaces that mark transitions between media.

```
<Handle surface interaction for path generation> ≡ 1006
    if (!foundIntersection) {
        <Capture escaped rays when tracing from the camera 1020>
        break;
    }
    <Compute scattering functions for mode and skip over medium boundaries 1007>
    <Initialize vertex with surface intersection information 1008>
    if (++bounces >= maxDepth)
        break;
    <Sample BSDF at current vertex and compute reverse probability 1008>
    ray = isect.SpawnRay(wi);
```

The fragment *<Capture escaped rays when tracing from the camera>* is necessary to support infinite area lights. It will be discussed in Section 16.3.5. The following fragment, *<Compute scattering functions for mode and skip over medium boundaries>*, is analogous to *<Compute scattering functions and skip over medium boundaries>* from the basic path tracer except that scattering functions are requested for the current light transport mode (radiance or importance transport) using the mode parameter.

```
Interaction::SpawnRay() 232
MediumInteraction::phase 688
PhaseFunction::Sample_p() 898
Sampler::Get2D() 422
SurfaceInteraction::bsdf 250
SurfaceInteraction::ComputeScatteringFunctions() 578
Vector3f 60
Vertex::CreateMedium() 997
```

```
<Compute scattering functions for mode and skip over medium boundaries> ≡ 1007
    isect.ComputeScatteringFunctions(ray, arena, true, mode);
    if (!isect.bsdf) {
        ray = isect.SpawnRay(ray.d);
        continue;
    }
```

Given a valid intersection, the current path vertex is initialized with the corresponding surface intersection vertex, where, again, the solid angle density `pdfFwd` is converted to an area density before being stored in `Vertex::pdfFwd`.

(Initialize vertex with surface intersection information) ≡ 1007
`vertex = Vertex::CreateSurface(isect, beta, pdfFwd, prev);`

If the maximum path depth has not yet been exceeded, a scattered direction is sampled from the BSDF and the path contribution in `beta` is updated. For the surface case, we can't generally assume that `BSDF::Pdf()` is symmetric; hence we must re-evaluate the sampling density with swapped arguments to obtain `pdfRev`. In case of a specular sampling event, we mark the vertex using the flag `Vertex::delta` and set `pdfFwd` and `pdfRev` to 0 to indicate that the underlying interaction has no continuous density function. Finally, we correct for non-symmetry related to the use of shading normals (see Section 16.1.3 for details).

(Sample BSDF at current vertex and compute reverse probability) ≡ 1007
`Vector3f wi, wo = isect.wo;`
`BxDFType type;`
`Spectrum f = isect.bsdf->Sample_f(wo, &wi, sampler.Get2D(), &pdfFwd,`
`BSDF_ALL, &type);`
`if (f.IsBlack() || pdfFwd == 0.f)`
`break;`
`beta *= f * AbsDot(wi, isect.shading.n) / pdfFwd;`
`pdfRev = isect.bsdf->Pdf(wi, wo, BSDF_ALL);`
`if (type & BSDF_SPECULAR) {`
`vertex.delta = true;`
`pdfRev = pdfFwd = 0;`
`}`
`beta *= CorrectShadingNormal(isect, wo, wi, mode);`

`AbsDot()` 64
`BSDF::Pdf()` 834
`BSDF::Sample_f()` 832
`BSDF_ALL` 513
`BSDF_SPECULAR` 513
`BxDFType` 513
`ConnectBDPT()` 1009
`CorrectShadingNormal()` 963
`Interaction::wo` 115
`Sampler::Get2D()` 422
`Spectrum` 315
`Spectrum::IsBlack()` 317
`SurfaceInteraction::bsdf` 250
`Vector3f` 60
`Vertex::ConvertDensity()`
`1000`
`Vertex::CreateSurface()` 997
`Vertex::delta` 998
`Vertex::pdfFwd` 1000
`Vertex::pdfRev` 1000

The loop wraps up by converting the reverse density `pdfRev` to a probability per unit area and storing it in the `Vertex` data structure of the preceding vertex.

(Compute reverse area density at preceding vertex) ≡ 1006
`prev.pdfRev = vertex.ConvertDensity(pdfRev, prev);`

16.3.3 SUBPATH CONNECTIONS

The `ConnectBDP()` function takes the light and camera subpaths and the number of vertices s and t to use from each one, respectively. It returns the corresponding strategy's contribution.

The connection strategy with $t = 1$ uses only a single camera vertex, the camera's position; the raster position of the path's contribution is then based on which pixel the last vertex of the light subpath is visible in (if any). In this case, the resulting position is returned via the `pRaster` argument.

(BDPT Method Definitions) ≡

```
Spectrum ConnectBDPT(const Scene &scene, Vertex *lightVertices,
    Vertex *cameraVertices, int s, int t,
    const Distribution1D &lightDistr, const Camera &camera,
    Sampler &sampler, Point2f *pRaster, Float *misWeightPtr) {
    Spectrum L(0.f);
    <Ignore invalid connections related to infinite area lights 1020>
    <Perform connection and write contribution to L 1009>
    <Compute MIS weight for connection strategy 1012>
    return L;
}
```

A number of cases must be considered when handling connections; special handling is needed for those involving short subpaths with only zero or one vertex. Some strategies dynamically sample an additional vertex, which is stored in the temporary variable sampled.

(Perform connection and write contribution to L) ≡ 1009

```
Vertex sampled;
if (s == 0) {
    <Interpret the camera subpath as a complete path 1009>
} else if (t == 1) {
    <Sample a point on the camera and connect it to the light subpath 1010>
} else if (s == 1) {
    <Sample a point on a light and connect it to the camera subpath>
} else {
    <Handle all other bidirectional connection cases 1011>
}
```

The first case ($s = 0$) applies when no vertices on the light subpath are used and can only succeed when the camera subpath p_0, p_1, \dots, p_{t-1} is already a complete path—that is, when vertex p_{t-1} can be interpreted as a light source. In this case, L is set to the product of the path throughput weight and the emission at p_{t-1} .

Camera 356
 Camera::Sample_Wi() 954
 Distribution1D 758
 Float 1062
 Point2f 68
 Sampler 421
 Scene 23
 Spectrum 315
 Vertex 996
 Vertex::beta 998
 Vertex::IsLight() 999
 Vertex::Le() 999

(Interpret the camera subpath as a complete path) ≡ 1009

```
const Vertex &pt = cameraVertices[t - 1];
if (pt.IsLight())
    L = pt.Le(scene, cameraVertices[t - 2]) * pt.beta;
```

The second case applies when $t = 1$ —that is, when a prefix of the light subpath is directly connected to the camera (Figure 16.15). To permit optimized importance sampling strategies analogous to direct illumination routines for light sources, we will ignore the actual camera vertex p_0 and sample a new one using Camera::Sample_Wi()—this optimization corresponds to the second bullet listed at the beginning of Section 16.3. This type of connection can only succeed if the light subpath vertex q_{s-1} supports sampled connections; otherwise the BSDF at q_{s-1} will certainly return 0 and there's no reason to attempt a connection.

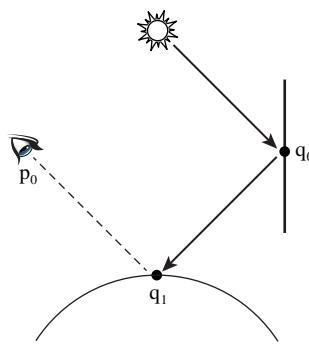


Figure 16.15: $t = 1$ Sampling Strategy for BDPT. We'd like to connect a subset of the light subpath to the camera. Given the last vertex of the light subpath, `Camera::Sample_Wi()` samples a vertex on the lens p_0 corresponding to a ray leaving the camera to the light path vertex (if there is such a ray that intersects the film).

(Sample a point on the camera and connect it to the light subpath) ≡

1009

```
const Vertex &qs = lightVertices[s - 1];
if (qs.IsConnectible()) {
    VisibilityTester vis;
    Vector3f wi;
    Float pdf;
    Spectrum Wi = camera.Sample_Wi(qs.GetInteraction(), sampler.Get2D(),
                                    &wi, &pdf, pRaster, &vis);
    if (pdf > 0 && !Wi.IsBlack()) {
        (Initialize dynamically sampled vertex and L for t = 1 case 1010)
    }
}
```

If the camera vertex was generated successfully, `pRaster` is initialized and `vis` holds the connection segment. Following Equation (16.1), we can compute the final contribution as the product of the subpath weights, the transmittance over the connecting segment, the BRDF or phase function, and a cosine factor when q_{s-1} is a surface vertex.

(Initialize dynamically sampled vertex and L for t = 1 case) ≡

1010

```
sampled = Vertex::CreateCamera(&camera, vis.P1(), wi / pdf);
L = qs.beta * qs.f(sampled) * vis.Tr(scene, sampler) * sampled.beta;
if (qs.IsOnSurface())
    L *= AbsDot(wi, qs.ns());
```

We omit the next case, $s = 1$, here. It corresponds to performing a direct lighting calculation at the last vertex of the camera subpath. Its implementation is similar to the $t = 1$ case—the main differences are that roles of lights and cameras are exchanged and that a light source must be chosen using `lightDistr` before a light sample can be generated.

The last case, *(Handle all other bidirectional connection cases)*, is responsible for most types of connections: it applies whenever the camera and light subpath prefixes are

AbsDot() 64
`Camera::Sample_Wi()` 954
`Float` 1062
`Sampler::Get2D()` 422
`Spectrum` 315
`Spectrum::IsBlack()` 317
`Vector3f` 60
`Vertex` 996
`Vertex::beta` 996
`Vertex::CreateCamera()` 997
`Vertex::f()` 998
`Vertex::GetInteraction()` 997
`Vertex::IsConnectible()` 998
`Vertex::IsOnSurface()` 998
`Vertex::ns()` 997
`VisibilityTester` 717
`VisibilityTester::P1()` 718
`VisibilityTester::Tr()` 718

long enough so that no special cases are triggered (i.e., when $s, t > 1$). If we consider the generalized path contribution equation from Section 15.5.1, we have constructed camera and light subpaths with the incremental path construction approach used in Section 14.5.3 for regular path tracing. Given the throughput of these paths up to the current vertices, $\hat{T}(\bar{q}_s)$ and $T(\bar{p}_t)$, respectively, where

$$\bar{p}_t = p_0, p_1, \dots, p_{t-1},$$

and similarly for \bar{q}_s , we can find that the contribution of a path of t light vertices and s camera vertices is given by

$$\begin{aligned}\hat{P}(\bar{q}_s \bar{p}_t) = L_e \hat{T}(\bar{q}_s) &\left[\hat{f}(q_{s-2} \rightarrow q_{s-1} \rightarrow p_{t-1}) \hat{G}(q_{s-1} \leftrightarrow p_{t-1}) \right. \\ &\left. \hat{f}(q_{s-1} \rightarrow p_{t-1} \rightarrow p_{t-2}) \right] \hat{T}(\bar{p}_t) W_e.\end{aligned}$$

The first and last products involving the emission, importance and generalized throughput terms, $L_e \hat{T}(\bar{q}_s)$ for the camera path and $\hat{T}(\bar{p}_t) W_e$ for the light path, are already available in the `Vertex::beta` fields of the connection vertices, so we only need to compute the term in brackets to find the path's overall contribution. The symmetric nature of BDPT is readily apparent: the final contribution is equal to the product of the subpath weights, the BRDF or phase functions and a (symmetric) generalized geometry term. Note that this strategy cannot succeed when one of the connection vertices is marked as not connectible—in this case, no connection attempt is made.

The product of subpath weights and the two BSDFs is often 0; this case happens, for example, if the connecting segment requires that light be transmitted through one of the two surfaces but the corresponding surface isn't transmissive. In this case, it's worth avoiding the unnecessary call to the `G()` function, which traces a shadow ray to test visibility.

```
(Handle all other bidirectional connection cases) ≡ 1009
AbsDot() 64
Float 1062
G() 1011
Sampler 421
Scene 23
Spectrum 315
Spectrum::IsBlack() 317
Vector3::LengthSquared() 65
Vector3f 60
Vertex 996
Vertex::beta 996
Vertex::f() 998
Vertex::GetInteraction() 997
Vertex::IsConnectible() 998
Vertex::IsOnSurface() 998
Vertex::ns() 997
Vertex::p() 997
VisibilityTester 717
VisibilityTester::Tr() 718

{Handle all other bidirectional connection cases} ≡ 1009
const Vertex &qs = lightVertices[s - 1], &pt = cameraVertices[t - 1];
if (qs.IsConnectible() && pt.IsConnectible()) {
    L = qs.beta * qs.f(pt) * pt.f(qs) * pt.beta;
    if (!L.IsBlack()) L *= G(scene, sampler, qs, pt);
}
}

{BDPT Utility Functions} +≡
Spectrum G(const Scene &scene, Sampler &sampler, const Vertex &v0,
           const Vertex &v1) {
    Vector3f d = v0.p() - v1.p();
    Float g = 1 / d.LengthSquared();
    d *= std::sqrt(g);
    if (v0.IsOnSurface())
        g *= AbsDot(v0.ns(), d);
    if (v1.IsOnSurface())
        g *= AbsDot(v1.ns(), d);
    VisibilityTester vis(v0.GetInteraction(), v1.GetInteraction());
    return g * vis.Tr(scene, sampler);
}
```

The generalized geometry term, Equation (15.5), is computed in a separate function `G()`.

```
{BDPT Utility Functions} +≡
Spectrum G(const Scene &scene, Sampler &sampler, const Vertex &v0,
           const Vertex &v1) {
    Vector3f d = v0.p() - v1.p();
    Float g = 1 / d.LengthSquared();
    d *= std::sqrt(g);
    if (v0.IsOnSurface())
        g *= AbsDot(v0.ns(), d);
    if (v1.IsOnSurface())
        g *= AbsDot(v1.ns(), d);
    VisibilityTester vis(v0.GetInteraction(), v1.GetInteraction());
    return g * vis.Tr(scene, sampler);
}
```

The computation of the multiple importance sampling weight for the connected path is implemented as a separate function `MISWeight()`, which we discuss next.

(Compute MIS weight for connection strategy) ≡ 1009

```
Float misWeight = L.IsBlack() ? 0.f :
    MISWeight(scene, lightVertices, cameraVertices, sampled, s, t,
              lightDistr);
L *= misWeight;
if (misWeightPtr) *misWeightPtr = misWeight;
```

16.3.4 MULTIPLE IMPORTANCE SAMPLING

Recall the example of a light pointed up at the ceiling, indirectly illuminating a room. Even without multiple importance sampling, bidirectional path tracing will do much better than path tracing by reducing the number of paths with no contribution, since the paths from the light provide camera path vertices more light-carrying targets to hit with connection segments (see Figure 16.17, which shows the effectiveness of various types of bidirectional connections). However, the image will still suffer from variance caused by paths with unexpectedly large contributions due to vertices on the camera subpaths that happen to find the bright spot on the ceiling. MIS can be applied to address this issue; it automatically recognizes that connection strategies that involve at least one scattering event on the light subpath lead to superior sampling strategies in this case. This ability comes at the cost of having to know the probabilities for constructing a path according to all available strategies and is the reason for caching the `Vertex::pdfFwd` and `Vertex::pdfRev` values earlier.

In this section, we will explain the `MISWeight()` function that computes the multiple importance sampling weight associated with a particular BDPT sampling strategy. It takes a light and camera subpath and an integer pair (s, t) identifying the prefixes used by a successful BDPT connection attempt, producing a complete path $q_0, \dots, q_{s-1}, p_{t-1}, \dots, p_0$. It iterates over all alternative strategies that could hypothetically have generated the same input path but with an earlier or later crossover point between the light and camera subpaths (Figure 16.16). The function then reweights the path contribution using the balance heuristic from Section 13.10.1, taking all of these possible sampling strategies into account.¹⁰ It is straightforward to switch to other MIS variants (e.g., based on the power heuristic) if desired.

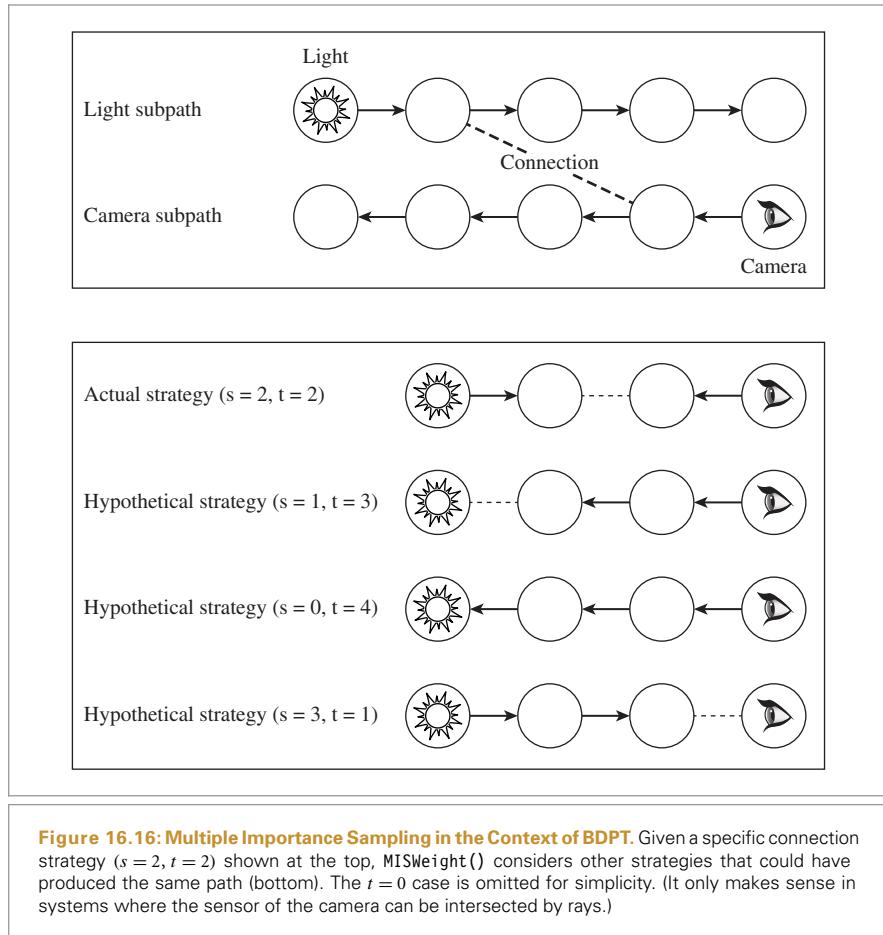
Let (s, t) denote the currently considered connection strategy, which concatenates a prefix q_0, \dots, q_{s-1} from the light subpath and a (reversed) prefix p_{t-1}, \dots, p_0 from the camera subpath, producing a path \bar{x} of length $n = s + t$ with vertices that we will refer to as x_i ($0 \leq i < n$):

$$\bar{x} = (x_0, \dots, x_{n-1}) = (q_0, \dots, q_{s-1}, p_{t-1}, \dots, p_0).$$

Suppose that the probability per unit area of vertex x_i is given by $p^-(x_i)$ and $p^+(x_i)$ for sampling strategies based on importance and radiance transport, respectively. Then the

Float [1062](#)
`MISWeight()` [1016](#)
`Spectrum::IsBlack()` [317](#)
`Vertex::pdfFwd` [1000](#)
`Vertex::pdfRev` [1000](#)

¹⁰ To keep the implementation simple, each vertex is assumed to be sampled from a 2D probability distribution. This may lead to slightly sub-optimal weights in participating media, where the distance sampling along rays causes the vertices to be distributed in three dimensions, though the method remains unbiased despite this inaccuracy.



area product density of the current path is simply the product of the importance transport densities up to vertex x_{s-1} and the radiance transport densities for the remainder:

$$p_s(\bar{x}) = p^{\rightarrow}(x_0) \cdots p^{\rightarrow}(x_{s-1}) \cdot p^{\leftarrow}(x_s) \cdots p^{\leftarrow}(x_{n-1}).$$

Implementation-wise, the above expression is straightforward to evaluate: the importance transport densities $p^{\rightarrow}(x_i)$ are already cached in the `Vertex::pdfFwd` fields of the light subpath, and the same holds true for the radiance transport densities on the camera subpath.

More generally, we are also interested in the path density according to other connection strategies (i, j) that could *in theory* have created this path. This requires that they generate paths of a compatible length—i.e., that $i + j = s + t$. Their corresponding path density is given by

$$p_i(\bar{x}) = p^{\rightarrow}(x_0) \cdots p^{\rightarrow}(x_{i-1}) \cdot p^{\leftarrow}(x_i) \cdots p^{\leftarrow}(x_{n-1}), \quad (16.17)$$

```
MISWeight() 1016
Vertex::pdfFwd 1000
Vertex::pdfRev 1000
```

where $0 \leq i \leq n$. Evaluating these will also involve the reverse probabilities in `Vertex::pdfRev`.



Figure 16.17: The Individual BDPT Strategies. Each row corresponds to light paths of a certain length. Note how almost every sampling strategy has deficiencies of some kind, evident in the form of high variance in these images. (Regular path tracing only samples the $s = 1$ paths.) Applying multiple importance sampling to path contributions is an effective way to reduce this variance.

Recall from Section 13.10.1 that the balance heuristic weight for strategy s out of a set of n sampling strategies with uniform sample allocation was given by

$$w_s(\bar{x}) = \frac{p_s(\bar{x})}{\sum_i p_i(\bar{x})}. \quad [16.18]$$

This is the expression we would like to evaluate in `MISWeight()`, though there are two practical issues that must first be addressed.

First, path densities can easily under- or overflow the range of representable values in single or even double precision. Area densities of individual vertices can be seen to be inversely proportional to the square of the scene dimensions: for instance, uniformly scaling the scene to half its size will quadruple the vertex densities. When computing the area product density of a path with 10 vertices, the same scaling operation will cause an increase in path density of approximately one million times. When working with very small or large scenes (compared to a box of unit volume), the floating point exponent of $p_i(\bar{x})$ can quickly exceed the valid range.

Second, a naive MIS implementation has time complexity of $O(n^4)$, where n is the maximum path length. Evaluation of $p_i(\bar{x})$ based on Equation (16.17) involves a linear sweep over n vertices, and the MIS weight in Equation (16.18) requires another sweep over n strategies. Given that this must be done once per connection strategy for a number of strategies that is proportional to the square of the subpath length, we are left with an algorithm of quartic complexity.

We will avoid both of these issues by using a more efficient incremental computation that works with ratios of probability densities to achieve better numerical and run-time behavior.



Figure 16.18: Variance Reduction Due to Multiple Importance Sampling. The same sampling strategies as in Figure 16.17, but now weighted using multiple importance sampling—effectively “turning off” each strategy where it does not perform well. The final result is computed by summing all of these images.

Dividing both the numerator and denominator of Equation (16.18) by $p_s(\bar{x})$ yields

$$w_s(\bar{x}) = \frac{1}{\sum_i \frac{p_i(\bar{x})}{p_s(\bar{x})}} = \left(\sum_{i=0}^{s-1} \frac{p_i(\bar{x})}{p_s(\bar{x})} + 1 + \sum_{i=s+1}^n \frac{p_i(\bar{x})}{p_s(\bar{x})} \right)^{-1} \quad (16.19)$$

The two sums above consider alternative strategies that would have taken additional steps on the camera or light subpath, respectively. Let us define a more concise notation for the individual summand terms:

$$r_i(\bar{x}) = \frac{p_i(\bar{x})}{p_s(\bar{x})}.$$

These satisfy the following recurrence relations:

$$\begin{aligned} r_i(\bar{x}) &= \frac{p_i(\bar{x})}{p_{i+1}(\bar{x})} \frac{p_{i+1}(\bar{x})}{p_s(\bar{x})} = \frac{p_i(\bar{x})}{p_{i+1}(\bar{x})} r_{i+1}(\bar{x}) \quad (i < s), \\ r_i(\bar{x}) &= \frac{p_i(\bar{x})}{p_{i-1}(\bar{x})} \frac{p_{i-1}(\bar{x})}{p_s(\bar{x})} = \frac{p_i(\bar{x})}{p_{i-1}(\bar{x})} r_{i-1}(\bar{x}) \quad (i > s). \end{aligned} \quad (16.20)$$

The recurrence weights in the above equations are ratios of path densities of two adjacent sampling strategies, which differ only in how a single vertex is generated. Thus, they can be reduced to probability ratios of the affected vertex:

$$\begin{aligned} \frac{p_i(\bar{x})}{p_{i+1}(\bar{x})} &= \frac{p^\rightarrow(x_0) \cdots p^\rightarrow(x_{i-1}) \cdot p^\leftarrow(x_i) \cdot p^\leftarrow(x_{i+1}) \cdots p^\leftarrow(x_{n-1})}{p^\rightarrow(x_0) \cdots p^\rightarrow(x_{i-1}) \cdot p^\rightarrow(x_i) \cdot p^\leftarrow(x_{i+1}) \cdots p^\leftarrow(x_{n-1})} = \frac{p^\leftarrow(x_i)}{p^\rightarrow(x_i)}, \\ \frac{p_i(\bar{x})}{p_{i-1}(\bar{x})} &= \frac{p^\rightarrow(x_0) \cdots p^\rightarrow(x_{i-2}) \cdot p^\rightarrow(x_{i-1}) \cdot p^\leftarrow(x_i) \cdots p^\leftarrow(x_{n-1})}{p^\rightarrow(x_0) \cdots p^\rightarrow(x_{i-2}) \cdot p^\leftarrow(x_{i-1}) \cdot p^\leftarrow(x_i) \cdots p^\leftarrow(x_{n-1})} = \frac{p^\rightarrow(x_{i-1})}{p^\leftarrow(x_{i-1})}. \end{aligned}$$

Combining this result with Equation (16.20), we obtain the following recursive expression for r_i :

$$r_i(\bar{x}) = \begin{cases} 1, & \text{if } i = s \\ \frac{p^{\leftarrow}(x_i)}{p^{\rightarrow}(x_i)} r_{i+1}(\bar{x}), & \text{if } i < s. \\ \frac{p^{\rightarrow}(x_{i-1})}{p^{\leftarrow}(x_{i-1})} r_{i-1}(\bar{x}), & \text{if } i > s. \end{cases} \quad [16.21]$$

The main portion of the `MISWeight()` function accumulates these probability ratios in a temporary variable `sumRi` using an incremental evaluation scheme based on Equation (16.21). The last line returns the reciprocal of the r_i terms according to Equation (16.19). There is also a special case at the beginning, which directly returns a weight of 1 for paths with two vertices, which can only be generated by a single strategy.

(BDPT Utility Functions) +≡

```
Float MISWeight(const Scene &scene, Vertex *lightVertices,
    Vertex *cameraVertices, Vertex &sampIed, int s, int t,
    const Distribution1D &lightPdf) {
    if (s + t == 2)
        return 1;
    Float sumRi = 0;
    (Define helper function remap0 that deals with Dirac delta functions 1016)
    (Temporarily update vertex properties for current strategy 1018)
    (Consider hypothetical connection strategies along the camera subpath 1017)
    (Consider hypothetical connection strategies along the light subpath 1017)
    return 1 / (1 + sumRi);
}
```

A helper function `remap0()` returns its argument while mapping 0-valued arguments to 1. It is used to handle the special case of Dirac delta functions in the path, which have a continuous density of 0. Such degenerate vertices cannot be joined using any deterministic connection strategy, and their discrete probability cancels when iterating over the remaining set of strategies because it occurs both in the numerator and denominator of the summands in Equation (16.19). The purpose of the helper function is to temporarily map these densities to a nonzero value to make sure that this cancellation occurs without causing a division by 0.

(Define helper function remap0 that deals with Dirac delta functions) ≡

1016

```
auto remap0 = [](float f) -> float { return f != 0 ? f : 1; };
```

To avoid an excessively large number of calls to the various `Vertex` PDF functions, the weight computation uses the cached probabilities in `Vertex::pdfFwd` and `Vertex::pdfRev`. Since these values only capture information about the original camera and light subpaths, they must still be updated to match the full path configuration near the cross-over point—specifically q_{s-1} and p_{t-1} and their predecessors. This is implemented in the somewhat technical fragment *(Temporarily update vertex properties for current strategy)*, which we discuss last.

We iterate over hypothetical strategies that would have taken additional steps from the light direction, using a temporary variable `ri` to store the current iterate r_i . The fragment

Distribution1D 758
 Float 1062
 MISWeight() 1016
 Scene 23
 Vertex 996
 Vertex::pdfFwd 1000
 Vertex::pdfRev 1000

name makes reference to the camera subpath, since these extra steps involve vertices that were in reality sampled from the camera side. All vertex densities are passed through the function `remap0()`, and the ratio is only added to a running sum when endpoints of the current hypothetical connection strategy are marked as non-degenerate. The loop terminates before reaching the $(n, 0)$ strategy, which shouldn't be considered since the camera cannot be intersected.

{Consider hypothetical connection strategies along the camera subpath} ≡ 1016

```
    Float ri = 1;
    for (int i = t - 1; i > 0; --i) {
        ri *= remap0(cameraVertices[i].pdfRev) /
            remap0(cameraVertices[i].pdfFwd);
        if (!cameraVertices[i].delta && !cameraVertices[i - 1].delta)
            sumRi += ri;
    }
```

The next step considers additional steps along the light subpath and largely resembles the previous case. A special case arises when the current strategy would involve intersecting a light source (i.e., when $s = 0$): this will fail when the endpoint involves a Dirac delta distribution, hence the additional test below.

{Consider hypothetical connection strategies along the light subpath} ≡ 1016

```
    ri = 1;
    for (int i = s - 1; i >= 0; --i) {
        ri *= remap0(lightVertices[i].pdfRev) /
            remap0(lightVertices[i].pdfFwd);
        bool deltaLightvertex = i > 0 ? lightVertices[i - 1].delta
                                         : lightVertices[0].IsDeltaLight();
        if (!lightVertices[i].delta && !deltaLightvertex)
            sumRi += ri;
    }
```

Finally, we will define the missing fragment *{Temporarily update vertex properties for current strategy}*, which modifies `Vertex` attributes with new values specific to the current connection strategy (s, t). To reduce the amount of code needed for both the update and the subsequent cleanup operations, we will introduce a helper class `ScopedAssignment` that temporarily modifies a given variable and then reverts its change when program execution leaves the scope it was defined in. It stores a pointer `ScopedAssignment::target` to a memory location of arbitrary type (specified via the `Type` template parameter) and a snapshot of the original value in `ScopedAssignment::backup`.

{BDPT Helper Definitions} +≡

```
template <typename Type> class ScopedAssignment {
public:
    {ScopedAssignment Public Methods 1018}
private:
    Type *target, backup;
};
```

Float 1062
 ScopedAssignment 1017
`ScopedAssignment::backup`
 1017
`ScopedAssignment::target`
 1017
`Vertex` 996
`Vertex::delta` 998
`Vertex::IsDeltaLight()` 999
`Vertex::pdfFwd` 1000
`Vertex::pdfRev` 1000

The `ScopedAssignment` constructor takes a pointer to a target memory location and overwrites it with the `value` parameter after making a backup copy. The destructor simply reverts any changes.

```
(ScopedAssignment Public Methods) ≡ 1017
  ScopedAssignment(Type *target = nullptr,
                    Type value = Type()) : target(target) {
    if (target) {
      backup = *target;
      *target = value;
    }
  }
  ~ScopedAssignment() { if (target) *target = backup; }
```

The main update operation then consists of finding the connection vertices and their predecessors and updating vertex probabilities and other attributes so that the two `Vertex` arrays reflect the chosen connection strategy.

```
(Temporarily update vertex properties for current strategy) ≡ 1016
  (Look up connection vertices and their predecessors 1018)
  (Update sampled vertex for s = 1 or t = 1 strategy 1018)
  (Mark connection vertices as non-degenerate 1019)
  (Update reverse density of vertex pt-1 1019)
  (Update reverse density of vertex pt-2 1019)
  (Update reverse density of vertices qs-1 and qs-2)
```

We begin by obtaining pointers to the affected connection vertices `qs-1` and `pt-1` and their predecessors.

```
(Look up connection vertices and their predecessors) ≡ 1018
  Vertex *qs      = s > 0 ? &lightVertices[s - 1] : nullptr,
  *pt      = t > 0 ? &cameraVertices[t - 1] : nullptr,
  *qsMinus = s > 1 ? &lightVertices[s - 2] : nullptr,
  *ptMinus = t > 1 ? &cameraVertices[t - 2] : nullptr;
```

Recall that strategies with $s = 1$ or $t = 1$ perform camera and light source sampling and thus generate a new endpoint. The implementation accounts for this by temporarily overriding `*qs` or `*pt` with the sampled vertex provided via the `sampled` argument of `MISWeight()`.

```
(Update sampled vertex for s = 1 or t = 1 strategy) ≡ 1018
  ScopedAssignment<Vertex> a1;
  if (s == 1)     a1 = { qs, sampled };
  else if (t == 1) a1 = { pt, sampled };
```

Certain materials in `pbrt` (e.g., the `UberMaterial`) instantiate both specular and non-specular `BxDF` lobes, which requires some additional consideration at this point: suppose the specular lobe of such a material is sampled while generating the camera or light subpath. In this case, the associated `Vertex` will have its `Vertex::delta` flag set to true, causing `MISWeight()` to (correctly) ignore it as a hypothetical connection vertex when comparing the densities of different strategies. On the other hand, it is possible that a

```
MISWeight() 1016
ScopedAssignment 1017
ScopedAssignment::backup
  1017
ScopedAssignment::target
  1017
UberMaterial 584
Vertex 996
Vertex::delta 998
```

BDPT strategy later connects this degenerate vertex to a vertex on the other subpath using its non-specular component. In this case, its “personality” must temporarily change to that of a non-degenerate vertex. We always force the `Vertex::delta` attribute of the connection vertices to `false` to account for this possibility.

(Mark connection vertices as non-degenerate) ≡

1018

```
ScopedAssignment<bool> a2, a3;
if (pt) a2 = { &pt->delta, false };
if (qs) a3 = { &qs->delta, false };
```

Next, we will update the reverse sampling densities of the connection vertices and their predecessors, starting with p_{t-1} . This vertex was originally sampled on the camera subpath, but it could also have been reached using an extra step from the light side ($q_{s-2} \rightarrow q_{s-1} \rightarrow p_{t-1}$ in three-point form); the resulting density at p_{t-1} is evaluated using `Vertex::Pdf()`.

The case $s = 0$ is special: here, p_t is an intersection with a light source found on the camera subpath. The alternative reverse sampling strategy generates a light sample using `Light::Sample_Le()`, and we evaluate its spatial density with the help of `Vertex::PdfLightOrigin()`.

(Update reverse density of vertex p_{t-1}) ≡

1018

```
ScopedAssignment<Float> a4;
if (pt)
    a4 = { &pt->pdfRev,
           s > 0 ? qs->Pdf(scene, qsMinus, *pt) :
                   pt->PdfLightOrigin(scene, *ptMinus, lightPdf) };
```

The next fragment initializes the `pdfRev` field of p_{t-2} with the density of the reverse strategy $q_{s-1} \rightarrow p_{t-1} \rightarrow p_{t-2}$. Once more, there is a special case for $s = 0$, where the alternative reverse strategy samples an emitted ray via `Light::Sample_Le()` and intersects it against the scene geometry; the corresponding density is evaluated using `Vertex::PdfLight()`.

(Update reverse density of vertex p_{t-2}) ≡

1018

```
ScopedAssignment<Float> a5;
if (ptMinus)
    a5 = { &ptMinus->pdfRev,
           s > 0 ? pt->Pdf(scene, qs, *ptMinus) :
                   pt->PdfLight(scene, *ptMinus) };
```

The last fragment, *(Update reverse density of vertices q_{s-1} and q_{s-2})*, is not included here. It is analogous except that it does not require a similar special case for $t = 0$.

16.3.5 INFINITE AREA LIGHTS AND BDPT

The infinite area light, first introduced in Section 12.6, provides a convenient way of illuminating scenes with realistic captured illumination. Unfortunately, its definition as an infinitely distant directional source turns out to be rather difficult to reconcile with BDPT’s path integral formulation, which expresses probabilities in terms of area densities, which in turn requires finite-sized emitting surfaces.

```
Float 1062
Light::Sample_Le() 955
ScopedAssignment 1017
Vertex::delta 998
Vertex::Pdf() 1001
Vertex::PdfLight() 1002
Vertex::PdfLightOrigin()
  1003
Vertex::pdfRev 1000
```

Through some gymnastics, we could represent infinite area lights with finite shapes. For example, an infinite area light's radiance emission distribution could be described by a large emitting sphere that surrounded the scene, where the directional distribution of emitted radiance at each point on the interior of the sphere was the same as the infinite area light's emitted radiance for the same direction. This approach would require a significantly more complex implementation of `InfiniteAreaLight` with no practical benefits apart from BDPT compatibility.

Instead of changing the functionality of `InfiniteAreaLight`, we will instead make infinite area lights a special case in BDPT. Since illumination from these lights is most naturally integrated over solid angles, our approach will be to add support for solid angle integration to the vertex abstraction layer. In practice, scenes may contain multiple infinite area lights; we will follow the convention of treating them as one combined light when evaluating the emitted radiance or determining sample probabilities.

First, we will create a special endpoint vertex any time a camera path ray escapes the scene. The *(Capture escaped rays when tracing from the camera)* fragment is invoked by `RandomWalk()` whenever no surface intersection could be found while generating the camera subpath. In the implementation of the `Vertex::CreateLight()` method called here, the `pdfFwd` variable recording the probability per unit solid angle is stored directly in `Vertex::pdfFwd` without conversion by `Vertex::ConvertDensity()`.

```
(Capture escaped rays when tracing from the camera) ≡ 1007
if (mode == TransportMode::Radiance) {
    vertex = Vertex::CreateLight(EndpointInteraction(ray), beta,
                                pdfFwd);
    ++bounces;
}
```

The existence of light vertices on the camera subpath leads to certain nonsensical connection strategies. For instance, we couldn't possibly connect a light vertex on the camera subpath to another vertex on the light subpath. The following check in `ConnectBDPT()` detects and ignores such connection attempts.

```
(Ignore invalid connections related to infinite area lights) ≡ 1009
if (t > 1 && s != 0 && cameraVertices[t - 1].type == VertexType::Light)
    return Spectrum(0.f);
```

Some parts of the code may still attempt to invoke `ConvertDensity()` with a `next` `Vertex` that refers to a infinite area light. The following fragment detects this case at the beginning of `ConvertDensity()` and directly returns the supplied solid angle density without conversion in that case.

```
(Return solid angle density if next is an infinite area light) ≡ 1000
if (next.IsInfiniteLight())
    return pdf;
```

Next, we need to adapt the light subpath sampling routine to correct the probability values returned by the ray sampling function `InfiniteAreaLight::Sample_Le()`. This case is detected in an additional fragment at the end of `GenerateLightSubpath()`.

```
ConnectBDPT() 1009
EndpointInteraction 996
GenerateLightSubpath() 1004
InfiniteAreaLight 737
InfiniteAreaLight::
    Sample_Le()
    959
Light 714
RandomWalk() 1005
Spectrum 315
TransportMode::Radiance 960
Vertex 996
Vertex::ConvertDensity()
    1000
Vertex::CreateLight() 997
Vertex::IsInfiniteLight()
    999
Vertex::type 996
VertexType::Light 996
```

```
{Correct subpath sampling densities for infinite area lights} ≡ 1005
    if (path[0].IsInfiniteLight()) {
        {Set spatial density of path[1] for infinite area light 1021}
        {Set spatial density of path[0] for infinite area light 1021}
    }
```

Recall that `InfiniteAreaLight::Sample_Le()` samples a ray direction (with a corresponding density `pdfDir`) and a ray origin on a perpendicular disk (with a corresponding density `pdfPos`) that touches the scene's bounding sphere. Due to foreshortening, the resulting ray has a corresponding spatial density of $\text{pdfPos} |\cos \theta|$ at its first intersection with the scene geometry, where θ is the angle between `ray.d` and the geometric normal.

```
{Set spatial density of path[1] for infinite area light} ≡ 1021
    if (nVertices > 0) {
        path[1].pdfFwd = pdfPos;
        if (path[1].IsOnSurface())
            path[1].pdfFwd *= AbsDot(ray.d, path[1].ng());
    }
```

Following our new convention, the spatial density of infinite area light endpoints is now expressed as a probability per unit solid angle. We will create a helper function `InfiniteLightDensity()` that determines this value while also accounting for the presence of other infinite area lights.

```
{Set spatial density of path[0] for infinite area light} ≡ 1021
    path[0].pdfFwd = InfiniteLightDensity(scene, lightDistr, ray.d);
```

```
AbsDot() 64
Distribution1D 758
Distribution1D::Count() 758
Distribution1D::func 758
Distribution1D::funcInt 758
Float 1062
InfiniteAreaLight::
    Sample_Le() 959
InfiniteLightDensity() 1021
Interaction 115
Light::flags 715
Light::Pdf_Li() 836
LightFlags::Infinite 715
Scene 23
Scene::lights 23
Vector3f 60
Vertex::IsInfiniteLight() 999
Vertex::IsOnSurface() 998
Vertex::ng() 997
Vertex::pdfFwd 1000
```

This function performs a weighted sum of the directional densities of all infinite area lights using the light probabilities in `lightDistr`.

```
{BDPT Helper Definitions} +≡
    inline Float InfiniteLightDensity(const Scene &scene,
        const Distribution1D &lightDistr, const Vector3f &w) {
        Float pdf = 0;
        for (size_t i = 0; i < scene.lights.size(); ++i)
            if (scene.lights[i]->flags & (int)LightFlags::Infinite)
                pdf += scene.lights[i]->Pdf_Li(Interaction(), -w) *
                    lightDistr.func[i];
        return pdf / (lightDistr.funcInt * lightDistr.Count());
```

The remaining two changes are analogous and address the probability computation in the `PdfLightOrigin()` and `PdfLight()` methods. For the former, we similarly return the combined solid angle density when an infinite area light is detected.

```
{Return solid angle density for infinite light sources} ≡ 1003
    return InfiniteLightDensity(scene, lightDistr, w);
```

In `PdfLight()`, we compute the probability of sampling a ray origin on a disk whose radius is equal to the scene's bounding sphere. The remainder of `PdfLight()` already

accounts for the necessary cosine foreshortening factor; hence we do not need to multiply by it here.

(Compute planar sampling density for infinite light sources) ≡ 1002

```
Point3f worldCenter;
Float worldRadius;
scene.WorldBound().BoundingSphere(&worldCenter, &worldRadius);
pdf = 1 / (Pi * worldRadius * worldRadius);
```

16.4 METROPOLIS LIGHT TRANSPORT

In 1997, Veach and Guibas proposed an unconventional rendering technique named *Metropolis Light Transport* (MLT), which applies the Metropolis-Hastings algorithm from Section 13.4 to the path space integral in Equation (16.1). Whereas all of the rendering techniques we have discussed until now have been based on the principles of Monte Carlo integration and independent sample generation, MLT adopts a different set of tools that allows samples to be *statistically correlated*.

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 path 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 admissible mutation rules: highly specialized mutations can be used to sample otherwise difficult-to-explore families of light-carrying paths in a way that would be tricky or impossible to realize without introducing bias in a standard Monte Carlo context.

The correlated nature of MLT provides an important advantage over methods based on independent sample generation, in that MLT is able to perform a *local exploration* of path space: when a path that makes a large contribution to the image is found, it's easy to find other similar paths by applying small perturbations to it (a sampling process that generates states based on the previous state value in this way is referred to as a *Markov chain*). The resulting short-term memory is often beneficial: when a function has a small value over most of its domain and a large contribution in only a small subset, local exploration amortizes the expense (in samples) of the search for the important region by taking many samples from this part of the path space. This property makes MLT a good choice for rendering particularly challenging scenes: while it is generally not able to match the performance of uncorrelated integrators 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.

The original MLT technique by Veach and Guibas builds on the path space theory of light transport, which presents additional challenges compared to the previous simple examples in Section 13.4.4: path space is generally not a Euclidean domain, as surface vertices are constrained to lie on 2D subsets of \mathbb{R}^3 . Whenever specular reflection or refraction occurs, a sequence of three adjacent vertices must satisfy a precise geometric relationship, which further reduces the available degrees of freedom.

Bounds3::BoundingSphere() 81

Float 1062

Pi 1063

Point3f 68

Scene::WorldBound() 24

MLT builds on a set of five mutation rules that each targets specific families of light paths. Three of the mutations perform a localized exploration of particularly challenging path classes such as caustics or paths containing sequences of specular-diffuse-specular interactions, while two perform larger steps with a corresponding lower overall acceptance rate. Implementing the full set of MLT mutations is a significant undertaking: part of the challenge is that none of the mutations is symmetric; hence an additional transition density function must be implemented for each one. Mistakes in any part of the system can cause subtle convergence artifacts that are notoriously difficult to debug.

16.4.1 PRIMARY SAMPLE SPACE MLT

In 2002, Kelemen et al. presented a rendering technique that is also based on the Metropolis-Hastings algorithm. We will refer to it as *Primary Sample Space MLT* (PSSMLT) for reasons that will become clear shortly. Like MLT, the PSSMLT method explores the space of light paths, searching for those that carry a significant amount of energy from a light source to the camera. The main difference is that PSSMLT does not use path space directly: it explores light paths indirectly, piggybacking on an existing Monte Carlo rendering algorithm such as uni- or bidirectional path tracing, which has both advantages and disadvantages. The main advantage is that PSSMLT can use symmetric transitions on a Euclidean domain, and for this reason it is much simpler to implement. The downside is that PSSMLT lacks detailed information about the structure of the constructed light paths, making it impossible to recreate the kinds of sophisticated mutation strategies that are found in the original MLT method.

The details of this approach are easiest to motivate from an implementation-centric viewpoint. Consider the case of rendering a scene with the path integrator from Section 14.5.4, where we generate (pseudo-)random samples in raster space to get a starting point on the film, turn them into rays, and invoke `PathIntegrator::Li()` to obtain corresponding radiance estimates. `PathIntegrator::Li()` will generally request a number of additional 1D or 2D samples from the Sampler to sample light sources and material models, perform Russian roulette tests, and so on. Conceptually, these samples could all be generated ahead of time and passed to `PathIntegrator::Li()` using an additional argument, thus completely removing any (pseudo-)randomness from an otherwise purely deterministic function.

Suppose that L is the resulting deterministic function, which maps an infinite-dimensional sample sequence X_1, X_2, \dots to a radiance estimate $L(X_1, X_2, \dots)$. Here, (X_1, X_2) denotes the raster position, and the remaining arguments X_3, X_4, \dots are the samples consumed by L .¹¹ By drawing many samples and averaging the results of multiple evaluations of L , we essentially compute high-dimensional integrals of L over a “hypercube” of (pseudo-)random numbers X_1, X_2, \dots .

Given this interpretation of integrating a radiance estimate function over a domain made of all possible sequences of input samples, we can apply the Metropolis-Hastings algorithm to create a sampling process on the same domain, with a distribution that is

`PathIntegrator::Li()` 876

Sampler 421

¹¹ 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.

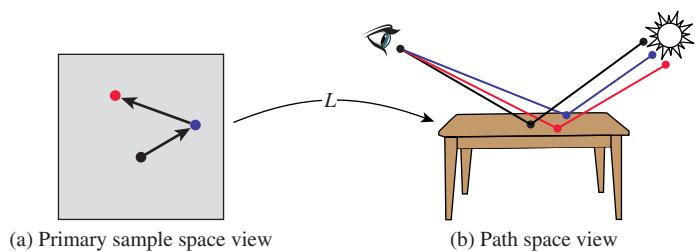


Figure 16.19: Primary Sample Space MLT. PSSMLT performs mutations in an abstract space of infinite dimensional “random number vectors” X . A deterministic mapping L constructs corresponding light paths on path space and estimates their radiance.

proportional to L (Figure 16.19). This distribution is intuitively desirable: more sampling effort is naturally placed in parts of the sampling space where more light transport occurs. The state space Ω for this problem consists of infinite dimensional sample vectors $(X_1, X_2, \dots) \in [0, 1]^\infty$ and is called the *primary sample space*. For brevity in the equations below, we will write the state vector as

$$X = (X_1, X_2, \dots).$$

We use the convention that all components (including the raster coordinates) are in the interval $[0, 1]$ and appropriately re-scaled within L when necessary.

PSSMLT explores primary sample space using two different kinds of mutations. The first (the “large step” mutation) replaces all components of the vector X with new uniformly distributed samples, which corresponds to invoking the underlying Monte Carlo rendering method as usual (i.e., without PSSMLT).

Recall from Section 13.4.2 that it is important that there be greater than zero probability that any possible sample value be proposed; this is taken care of by the large step mutation. In general, the large step mutation helps us 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 X_i . This mutation explores light-carrying paths close to the current path, which is particularly important when a difficult lighting configuration is encountered.

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).

The interface between the outer Metropolis-Hastings iteration and the inner Integrator only involves the exchange of abstract sample vectors, making this an extremely general approach: PSSMLT can theoretically enhance any kind of rendering method that is based on Monte Carlo integration—in fact, it even works for general Monte Carlo integration problems that are not related to rendering at all.

In practice, PSSMLT is often implemented on top of an existing bidirectional path tracer. The resulting algorithm generates a new primary sample space state in every iteration and passes it to BDPT, which invokes its set of connection strategies and re-weights the result using MIS. In this setting, mutations effectively jump from one group of path connections to another group rather than dealing with individual connection strategies. However, this is not without disadvantages: in many cases, only a small subset of the strategies is truly effective, and MIS will consequently assign a large weight only to this subset. This means that the algorithm still spends a considerable portion of its time generating connections with strategies that have a low weight and thus contribute very little to the rendered image.

16.4.2 MULTIPLEXED MLT

In 2014, Hachisuka et al. presented an extension to PSSMLT named *Multiplexed Metropolis Light Transport* (MMLT) to address this problem. MMLT leaves the “outer” Metropolis-Hastings iteration conceptually unchanged and applies a small but effective modification to the “inner” BDPT integrator. Instead of always invoking all BDPT connection strategies, the algorithm chooses a single strategy according to an extra state dimension and returns its contribution scaled by the inverse discrete probability of the choice. The additional dimension used for strategy selection can be mutated using small or large steps in the same way as the other primary sample space components of X.

To prevent unintentional large structural path mutations, Hachisuka et al. fix the Markov chain so that it only explores paths of a fixed depth value; the general light transport problem is then handled by running many independent Markov chains.

The practical consequence is that the Metropolis sampler will tend to spend more computation on effective strategies that produce larger MIS-weighted contributions to the image. Furthermore, the individual iterations are much faster since they only involve a single connection strategy. The combination of these two aspects improves the Monte Carlo efficiency of the resulting estimator.

Figure 16.20 shows the contemporary house scene rendered with bidirectional path tracing and MMLT, using roughly equal computation time for each approach, and Figure 16.21 compares them with the San Miguel scene. For both, MMLT 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 16.1 helps illustrate these efficiency differences: for both of these scenes, both path tracing and BDPT have a lot of trouble finding paths that carry any radiance, while Metropolis is more effective at doing so thanks to path reuse.

16.4.3 APPLICATION TO RENDERING

Metropolis sampling generates samples from the distribution of a given scalar function. To apply it to rendering, there are two issues that we must address: first, we need to estimate a separate integral for each pixel to turn the generated samples into an image, and, second, we need to handle the fact that L is a spectrally valued function but Metropolis needs a scalar function to compute the acceptance probability, Equation (13.7).



(a)



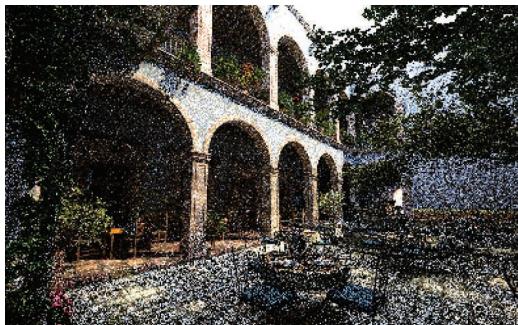
(b)

Figure 16.20: Comparison of Bidirectional Path Tracing and Multiplexed Metropolis Light Transport. (a) Rendered with bidirectional path tracing with 128 samples per pixel. Even with many samples, the image is quite noisy. (b) Rendered with Multiplexed Metropolis Light Transport with an average of 420 mutations per pixel (roughly equal running time). MLT generates a substantially better image for the same amount of work. BDPT does do better in a few localized parts of the image: note the directly illuminated surfaces outside the windows, for example. There, the ability to use well-distributed sample points for the direct lighting calculation gives a result with lower variance. (*Model courtesy of Florent Boyer.*)

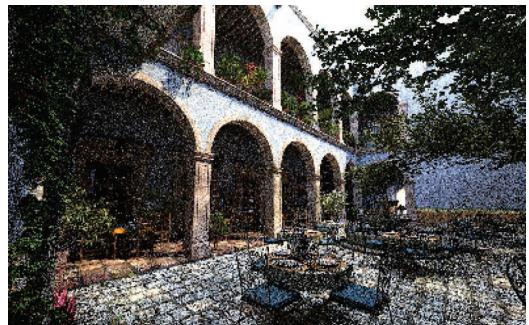
We can apply the ideas from Section 13.4.5 to use Metropolis samples to compute integrals. First, we define the *image contribution function* such that for an image with j pixels, each pixel I_j has a value that is the integral of the product of the pixel's image reconstruction filter h_j and the radiance L that contributes to the image:

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

The filter function h_j only depends on the two components of \mathbf{X} associated with the raster position. The value of h_j for any particular pixel is usually 0 for the vast majority of samples \mathbf{X} due to the filter's finite extent.



(a)



(b)



(c)

Figure 16.21: Comparison of Path Tracing, BDPT, and Multiplexed Metropolis Light Transport. (a) Path tracing with 200 samples per pixel. (b) Bidirectional path tracing with 128 samples per pixel. (c) Metropolis Light Transport with an average of 950 mutations per pixel (roughly equal running time). BDPT is notably more effective than path tracing, and as with Figure 16.20, MLT is the most effective. (*Model courtesy of Guillermo M. Leal Llaguno.*)

Table 16.1: Percentage of Traced Paths That Carried Zero Radiance. With these two scenes, both path tracing and BDPT have trouble finding light-carrying paths: the vast majority of the generated paths don't carry 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 those approaches here.

	Path tracing	BDPT	MMLT
Modern House	98.0%	97.6%	51.9%
San Miguel	95.9%	97.0%	62.0%

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

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

Recall that Metropolis sampling requires a scalar function that defines the desired distribution of samples generated by the algorithm. Unfortunately, L is a spectrally valued function, and thus there is no unambiguous notion of what it means to generate samples proportional to L . To work around this issue, we will define a *scalar contribution function* $I(\mathbf{X})$ that is used inside the Metropolis iteration. 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 :

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

and the pixel values can thus be computed as

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

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

$$I_j \approx \frac{b}{N} \sum_{i=1}^N \frac{h_j(\mathbf{X}_i) L(\mathbf{X}_i)}{I(\mathbf{X}_i)}. \quad (16.22)$$

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 $L_i(\mathbf{X}_i)/I(\mathbf{X}_i)$ is generally of the same magnitude).

16.4.4 PRIMARY SAMPLE SPACE SAMPLER

The MLTIntegrator applies Metropolis sampling and MMLT to render images, using the bidirectional path tracer from Section 16.3. Its implementation is contained in the files

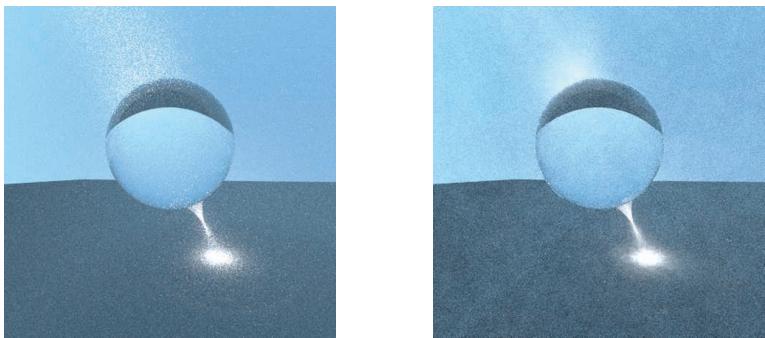


Figure 16.22: A Volumetric Caustic. Light passing through the sphere is focused in the medium behind it, creating a volumetric caustic. (a) Rendered with bidirectional path tracing, (b) rendered with the MLTIntegrator, using a roughly equal amount of time. Note that MMLT gives a lower variance result, thanks to being able to efficiently explore the local path space once a high-contribution path has been found.

integrators/mlt.h and integrators/mlt.cpp. Figure 16.22 shows the effectiveness of this integrator with a particularly tricky lighting situation. Before describing its implementation, we'll first introduce the MLTSampler, which is responsible for managing primary sample space state vectors, mutations, and acceptance and rejection steps.

```
(MLTSampler Declarations) ≡
class MLTSampler : public Sampler {
public:
    (MLTSampler Public Methods 1030)
protected:
    (MLTSampler Private Declarations 1030)
    (MLTSampler Private Methods)
    (MLTSampler Private Data 1030)
};
```

The MLTIntegrator works best if the MLTSampler actually maintains three separate sample vectors—one for the camera subpath, one for the light subpath, and one for the connection step. We'll say that these are three *sample streams*. The streamCount parameter to the constructor lets the caller request a particular number of such sample streams.

Later, during the initialization phase of MLTIntegrator, we will create many separate MLTSampler instances that are used to select a suitable set of starting states for the Metropolis sampler. Importantly, this process requires that each MLTSampler produces a distinct sequence of state vectors. The RNG pseudo-random number generator used by pbrt has a handy feature that makes this easy to accomplish: the RNG constructor accepts a *sequence index*, which selects between one of 2^{63} unique pseudo-random sequences. We thus add an rngSequenceIndex parameter to the MLTSampler constructor that is used to supply a unique stream index to the internal RNG.

MLTIntegrator 1035

MLTSampler 1029

RNG 1065

Sampler 421

(MLTSampler Public Methods) ≡ 1029

```
MLTSampler(int mutationsPerPixel, int rngSequenceIndex,
           Float sigma, Float largeStepProbability, int streamCount)
: Sampler(mutationsPerPixel), rng(rngSequenceIndex), sigma(sigma),
  largeStepProbability(largeStepProbability),
  streamCount(streamCount) { }
```

The `largeStepProbability` parameter refers to the probability of taking a “large step” mutation, and `sigma` controls the size of “small step” mutations.

(MLTSampler Private Data) ≡ 1029

```
RNG rng;
const Float sigma, largeStepProbability;
const int streamCount;
```

The `MLTSampler::X` member variable stores the current sample vector `X`. Because we generally don’t know ahead of time how many dimensions of `X` are actually needed during the sampler’s lifetime, we’ll start with an empty vector and expand it on demand as calls to `MLTSampler::Get1D()` and `MLTSampler::Get2D()` occur during rendering.

(MLTSampler Private Data) +≡ 1029
`std::vector<PrimarySample> X;`

The elements of this array have the type `PrimarySample`. The main task for `PrimarySample` is to record the current value of a single component of `X` on the interval [0, 1). In the following, we’ll add some additional functionality for representing proposed mutations and restoring the original sample value if a proposed mutation is rejected.

(MLTSampler Private Declarations) ≡ 1029

```
struct PrimarySample {
    Float value = 0;
(PrimarySample Public Methods 1034)
(PrimarySample Public Data 1032)
};
```

The `Get1D()` method returns the value of a single component of `MLTSampler::X`, whose position is given by `GetNextIndex()`—for now, we can think of this method as returning the value of a running counter that increases with every call. `EnsureReady()` expands `MLTSampler::X` as needed and ensures that its contents are in a consistent state. The details of these methods will be clearer after a few more preliminaries, so we won’t introduce their implementations just yet.

(MLTSampler Method Definitions) ≡

```
Float MLTSampler::Get1D() {
    int index = GetNextIndex();
    EnsureReady(index);
    return X[index].value;
}
```

The 2D analog simply performs two calls to `Get1D()`.

```
Float 1062
MLTSampler 1029
MLTSampler::EnsureReady() 1032
MLTSampler::Get1D() 1030
MLTSampler::Get2D() 1031
MLTSampler::GetNextIndex() 1035
MLTSampler::largeStepProbability 1030
MLTSampler::rng 1030
MLTSampler::sigma 1030
MLTSampler::streamCount 1030
MLTSampler::X 1030
PrimarySample 1030
PrimarySample::value 1030
RNG 1065
Sampler 421
```

```
{MLTSampler Method Definitions} +≡
    Point2f MLTSampler::Get2D() {
        return Point2f(Get1D(), Get1D());
    }
```

Next, we will define several `MLTIntegrator` methods that are not part of the official Sampler interface. The first one, `MLTSampler::StartIteration()`, is called at the beginning of each Metropolis-Hastings iteration; it increases the `currentIteration` counter and determines which type of mutation (small or large) should be applied to the sample vector in the current iteration.

```
{MLTSampler Method Definitions} +≡
void MLTSampler::StartIteration() {
    currentIteration++;
    largeStep = rng.UniformFloat() < largeStepProbability;
}
```

`currentIteration` is a running counter, which keeps track of the current Metropolis-Hastings iteration index. Note that iterations with rejected proposals will be excluded from this count.

```
{MLTSampler Private Data} +≡
int64_t currentIteration = 0;
bool largeStep = true;
```

1029

The `MLTSampler::lastLargeStepIteration` member variable records the index of the last iteration where a successful large step took place. Our implementation chooses the initial state X_0 to be uniformly distributed on primary sample space $[0, 1)^\infty$; hence the first iteration's state can be interpreted as the result of a large step in iteration 0.

```
MLTIntegrator 1035
MLTSampler 1029
MLTSampler::Accept() 1031
MLTSampler::currentIteration
    1031
MLTSampler::EnsureReady()
    1032
MLTSampler::Get1D() 1030
MLTSampler::largeStep 1031
MLTSampler::
    LargeStepProbability
    1030
MLTSampler::
    lastLargeStepIteration
    1031
MLTSampler::rng 1030
MLTSampler::StartIteration()
    1031
MLTSampler::X 1030
Point2f 68
RNG::UniformFloat() 1066
Sampler 421
```

```
{MLTSampler Private Data} +≡
int64_t lastLargeStepIteration = 0;
```

1029

The `MLTSampler::Accept()` method is called whenever a proposal is accepted.

```
{MLTSampler Method Definitions} +≡
void MLTSampler::Accept() {
    if (largeStep)
        lastLargeStepIteration = currentIteration;
}
```

At this point, the main missing parts are `MLTSampler::EnsureReady()` and the logic that applies the actual mutations to `MLTSampler::X`. Before filling in those gaps, let us take a brief step back.

In theory, all entries in the `X` vector must be updated by small or large mutations in every iteration of the Metropolis sampler. However, doing so would sometimes be rather inefficient: consider the case where most iterations only query a small number of dimensions of `X`, hence `MLTSampler::X` has not grown to a large size yet. If a later iteration makes many calls to `Get1D()` or `Get2D()`, then the dynamic array `MLTSampler::X` must correspondingly expand, and these extra entries increase the cost of every subsequent Metropolis iteration (even if these components of `X` are never referenced again!).

Instead, it's more efficient to update the PrimarySample entries on demand—that is, when they are referenced by an actual Get1D() or Get2D() call. Doing so avoids the aforementioned inefficiency, though after some period of inactivity, we must carefully replay all mutations that a given PrimarySample missed. To keep track of this information, an additional member variable recording the last iteration where a PrimarySample was modified is useful.

```
(PrimarySample Public Data) ≡ 1030
    int64_t lastModificationIteration = 0;
```

We now have enough background to proceed to the implementation of the MLTSampler::EnsureReady() method, which updates individual sample values when they are accessed.

```
(MLTSampler Method Definitions) +≡
void MLTSampler::EnsureReady(int index) {
    {Enlarge MLTSampler::X if necessary and get current Xi 1032}
    {Reset Xi if a large step took place in the meantime 1032}
    {Apply remaining sequence of mutations to sample 1033}
}
```

First, any gaps are filled with zero-initialized PrimarySamples before a reference to the requested entry is obtained.

```
{Enlarge MLTSampler::X if necessary and get current Xi) ≡ 1032
if (index >= X.size())
    X.resize(index + 1);
PrimarySample &Xi = X[index];
```

When the last modification of X_i precedes the last large step, the current content of $X_i.value$ is irrelevant, since it should have been overwritten with a new uniform sample in iteration `lastLargeStepIteration`. In this case, we simply replay this missed mutation and update the last modification iteration index accordingly.

```
{Reset Xi if a large step took place in the meantime} ≡ 1032
if (Xi.lastModificationIteration < lastLargeStepIteration) {
    Xi.value = rng.UniformFloat();
    Xi.lastModificationIteration = lastLargeStepIteration;
}
```

Next, a call to `Backup()` notifies the PrimarySample that a mutation is going to be proposed; doing so allows it to make a copy of X_i 's sample value in case the mutation is rejected. All remaining mutations between iterations `lastLargeStepIteration` and `currentIteration` are then applied. Two different cases can occur: when the current iteration is a large step, we simply initialize PrimarySample::value with a uniform sample. Otherwise, all iterations since the last large step are (by definition) small steps that we must replay.

```
MLTSampler 1029
MLTSampler::EnsureReady()
    1032
    MLTSampler::
        lastLargeStepIteration
            1031
    MLTSampler::X 1030
    PrimarySample 1030
    PrimarySample::
        lastModificationIteration
            1032
    PrimarySample::value 1030
    RNG::UniformFloat() 1066
```

```

⟨Apply remaining sequence of mutations to sample⟩ ≡ 1032
    Xi.Backup();
    if (largeStep) {
        Xi.value = rng.UniformFloat();
    } else {
        int64_t nSmall = currentIteration - Xi.lastModificationIteration;
        ⟨Apply nSmall small step mutations 1033⟩
    }
    Xi.lastModificationIteration = currentIteration;

```

For small steps, we apply normally distributed perturbations to each component:

$$X'_i \sim N(X_i, \sigma^2),$$

where σ is given by `MLTIntegrator::sigma`. The advantage of sampling with a normal distribution like this is that it naturally tries a variety of mutation sizes. It preferentially makes small mutations that remain close to the current state, 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.

Our implementation here merges the sequence of `nSmall` perturbations into a single update and clamps the result to the unit interval, wrapping around to the other end of the domain if necessary. Wrapping ensures that the transition probabilities for all pairs of sample values are symmetric.

```

⟨Apply nSmall small step mutations⟩ ≡ 1033
    ⟨Sample the standard normal distribution N(0, 1) 1034⟩
    ⟨Compute the effective standard deviation and apply perturbation to Xi 1034⟩

```

Recall the rule that when two samples from normal distributions $N(\mu_1, \sigma_1^2)$ and $N(\mu_2, \sigma_2^2)$ are added, the sum is also normally distributed with parameters $N(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2)$. Thus, when n perturbations are to be applied to X_i , instead of performing n perturbations in sequence, it is equivalent and more efficient to directly sample

$$X'_i \sim N(X_i, n\sigma^2), \quad (16.23)$$

which we do by importance sampling a standard normal distribution and scaling the result by $\sqrt{n}\sigma$. Applying the inversion method to the PDF

$$p(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

gives the following sampling method for a uniform sample $\xi \in [0, 1]$:

$$P^{-1}(\xi) = \sqrt{2} \operatorname{erf}^{-1}(2\xi - 1),$$

```

MLTIntegrator::sigma 1039
MLTSampler::currentIteration
1031
MLTSampler::largeStep 1031
PrimarySample::Backup() 1034
PrimarySample::
lastModificationIteration
1032
PrimarySample::value 1030
RNG::UniformFloat() 1066

```

where erf is the error function, $\operatorname{erf}(x) = 2/\sqrt{\pi} \int_0^x e^{-x'^2} dx'$, and erf^{-1} is its inverse. The `ErfInv()` function, not included here, approximates erf^{-1} with a polynomial.

(Sample the standard normal distribution $N(0, 1)$) ≡ 1033

```
Float normalSample = Sqrt2 * ErfInv(2 * rng.UniformFloat() - 1);
```

We scale the resulting sample by the effective variance from Equation (16.23) and use it to perturb the sample X_i before keeping only its fractional component, so that it remains in $[0, 1]$.

(Compute the effective standard deviation and apply perturbation to X_i) ≡ 1033

```
Float effSigma = sigma * std::sqrt((Float)nSmall);
Xi.value += normalSample * effSigma;
Xi.value -= std::floor(Xi.value);
```

`MLTSampler::Reject()` must be called whenever a proposed mutation is rejected. It restores all `PrimarySamples` modified in the current iteration and reverts the iteration counter.

(MLTSampler Method Definitions) +≡

```
void MLTSampler::Reject() {
    for (auto &Xi : X)
        if (Xi.lastModificationIteration == currentIteration)
            Xi.Restore();
    --currentIteration;
}
```

The `Backup()` and `Restore()` methods make it possible to record the value of a `PrimarySample` before a mutation and to restore it if the mutation is rejected.

(PrimarySample Public Methods) ≡ 1030

```
void Backup() {
    valueBackup = value;
    modifyBackup = lastModificationIteration;
}
void Restore() {
    value = valueBackup;
    lastModificationIteration = modifyBackup;
}
```

(PrimarySample Public Data) +≡ 1030

```
Float valueBackup = 0;
int64_t modifyBackup = 0;
```

Before wrapping up `MLTSampler`, we must address a detail that would otherwise cause issues when the sampler is used with `BDPT`. For each pixel sample, the `BDPTIntegrator` implementation calls `GenerateCameraSubpath()` and `GenerateLightSubpath()` in sequence to generate a pair of subpaths, each function requesting 1D and 2D samples from a supplied `Sampler` as needed.

In the context of `MLT`, the resulting sequence of sample requests creates a mapping between components of X and vertices on the camera or light subpath. With the process described above, the components X_0, \dots, X_n determine the camera subpath (for some $n \in \mathbb{Z}$), and the remaining values X_{n+1}, \dots, X_m determine the light subpath. If the

```
BDPTIntegrator 992
Float 1062
GenerateCameraSubpath() 1003
GenerateLightSubpath() 1004
MLTSampler 1029
MLTSampler::currentIteration 1031
MLTSampler::Reject() 1034
MLTSampler::sigma 1030
MLTSampler::X 1030
PrimarySample 1030
PrimarySample::lastModificationIteration 1032
PrimarySample::modifyBackup 1034
PrimarySample::Restore() 1034
PrimarySample::value 1030
PrimarySample::valueBackup 1034
RNG::UniformFloat() 1066
Sampler 421
Sqrt2 1063
```

camera subpath requires a different number of samples after a perturbation (e.g., because the random walk produced fewer vertices), then there is a shift in the assignment of primary sample space components to the light subpath. This leads to an unintended large-scale modification to the light path.

It is easy to avoid this problem with a more careful indexing scheme: the `MLTSampler` partitions `X` into multiple interleaved streams that cannot interfere with each other. The `MLTSampler::StartStream()` method indicates that subsequent samples should come from the stream with the given index. It also resets `sampleIndex`, the index of the current sample in the stream. (The number of such streams, `MLTSampler::streamCount`, is specified in the `MLTSampler` constructor.)

```
{MLTSampler Method Definitions} +≡
void MLTSampler::StartStream(int index) {
    streamIndex = index;
    sampleIndex = 0;
}
```

```
{MLTSampler Private Data} +≡
int streamIndex, sampleIndex;
```

1029

After the stream is selected, the `MLTSampler::GetNextIndex()` method performs corresponding steps through the primary sample vector components. It interleaves the streams into the global sample vector—in other words, the first `streamCount` dimensions in `X` are respectively used for the first dimension of each of the streams, and so forth.

```
{MLTSampler Public Methods} +≡
int GetNextIndex() {
    return streamIndex + streamCount * sampleIndex++;
}
```

1029

16.4.5 MLT INTEGRATOR

Given all of this infrastructure—an explicit representation of an n -dimensional sample `X`, functions to apply mutations to it, and BDPT’s vertex abstraction layer for evaluating the radiance of a given sample value—we can move forward to the heart of the implementation of the `MLTIntegrator`.

```
{MLT Declarations} ≡
Integrator 25
MLTSampler 1029
MLTSampler::GetNextIndex() 1035
MLTSampler::sampleIndex 1035
MLTSampler::StartStream() 1035
MLTSampler::streamCount 1030
MLTSampler::streamIndex 1035

class MLTIntegrator : public Integrator {
public:
    {MLTIntegrator Public Methods}
private:
    {MLTIntegrator Private Data 1037}
};
```

The `MLTIntegrator` constructor, not shown here, just initializes various member variables from parameters provided to it. These member variables will be introduced in the following as they are used.

We begin by defining the method `MLTIntegrator::L()`, which computes the radiance $L(X)$ for a vector of sample values X provided by an `MLTSampler`. Its parameter `depth` specifies a specific path depth, and `pRaster` returns the raster position of the path, if the path successfully carries light from a light source to the film plane. The initial statement activates the first of three streams in the underlying `MLTSampler`.

(MLT Method Definitions) ≡

```
Spectrum MLTIntegrator::L(const Scene &scene, MemoryArena &arena,
    const std::unique_ptr<Distribution1D> &lightDistr,
    MLTSampler &sampler, int depth, Point2f *pRaster) {
    sampler.StartStream(cameraStreamIndex);
    (Determine the number of available strategies and pick a specific one 1036)
    (Generate a camera subpath with exactly t vertices 1037)
    (Generate a light subpath with exactly s vertices 1037)
    (Execute connection strategy and return the radiance estimate 1037)
}
```

The implementation uses three sample streams from the `MLTSampler`: the first two for the camera and light subpath and the third one for any `Camera::Sample_Wi()` or `Light::Sample_Li()` calls performed by connection strategies in `ConnectBDPT()` with $s = 1$ or $t = 1$ (refer to Section 16.3.3 for details).

(MLTSampler Constants) ≡

```
static const int cameraStreamIndex = 0;
static const int lightStreamIndex = 1;
static const int connectionStreamIndex = 2;
static const int nSampleStreams = 3;
```

The body of `MLTIntegrator::L()` first selects an individual BDPT strategy for the provided `depth` value—this is the MMLT modification to PSSMLT—and invokes the bidirectional path-tracing machinery to compute a corresponding radiance estimate. For paths with zero scattering events (i.e., directly observed light sources), the only viable strategy provided by the underlying BDPT implementation entails intersecting them with a ray traced from the camera. For longer paths, there are $depth + 2$ possible strategies. The fragment below uniformly maps the first primary sample space dimension onto this set of strategies. The variables s and t denote the number of light and camera subpath sampling steps following the convention of the `BDPTIntegrator`.

(Determine the number of available strategies and pick a specific one) ≡

```
int s, t, nStrategies;
if (depth == 0) {
    nStrategies = 1;
    s = 0;
    t = 2;
} else {
    nStrategies = depth + 2;
    s = std::min((int)(sampler.Get1D() * nStrategies), nStrategies - 1);
    t = nStrategies - s;
}
```

<code>BDPTIntegrator</code> 992 <code>Camera::Sample_Wi()</code> 954 <code>ConnectBDPT()</code> 1009 <code>Distribution1D</code> 758 <code>Light::Sample_Li()</code> 716 <code>MemoryArena</code> 1074 <code>MLTIntegrator</code> 1035 <code>MLTIntegrator::cameraStreamIndex</code> <code>1036</code>	<code>MLTIntegrator::L()</code> 1036 <code>MLTSampler</code> 1029 <code>MLTSampler::Get1D()</code> 1030 <code>MLTSampler::StartStream()</code> <code>1035</code> <code>Point2f</code> 68 <code>Scene</code> 23 <code>Spectrum</code> 315
--	---

The next three fragments compute the radiance estimate. They strongly resemble BDPT with some MMLT-specific modifications: the first one samples a film position in $[0, 1]^2$, maps it to raster coordinates, and tries to generate a corresponding camera subpath with *exactly t* vertices, failing with a 0-valued estimate for L when this was not possible.

```
(Generate a camera subpath with exactly t vertices) ≡ 1036
    Vertex *cameraVertices = arena.Alloc<Vertex>(t);
    Bounds2f sampleBounds = (Bounds2f)camera->film->GetSampleBounds();
    *pRaster = sampleBounds.Lerp(sampler.Get2D());
    if (GenerateCameraSubpath(scene, sampler, arena, t, *camera,
                                *pRaster, cameraVertices) != t)
        return Spectrum(0.f);
```

The camera member variable holds the Camera specified in the scene description file.

```
{MLTIntegrator Private Data} ≡ 1035
    std::shared_ptr<const Camera> camera;
```

The next fragment implements an analogous operation for the light subpath. Note the call to MLTSampler::StartStream(), which switches to the second stream of samples.

```
(Generate a light subpath with exactly s vertices) ≡ 1036
    sampler.StartStream(lightStreamIndex);
    Vertex *lightVertices = arena.Alloc<Vertex>(s);
    if (GenerateLightSubpath(scene, sampler, arena, s,
                            cameraVertices[0].time(),
                            *lightDistr, lightVertices) != s)
        return Spectrum(0.f);

Bounds2::Lerp() 80
Bounds2f 76
Camera 356
Camera::film 356
ConnectBDPT() 1009
Film::GetSampleBounds() 487
GenerateCameraSubpath() 1003
GenerateLightSubpath() 1004
MemoryArena::Alloc() 1074
MLTIntegrator::camera 1037
MLTIntegrator::
    connectionStreamIndex 1036
MLTIntegrator::
    lightStreamIndex 1036
MLTIntegrator::Render() 1038
MLTSampler::StartStream() 1035
Sampler::Get2D() 422
Spectrum 315
Vertex 996
Vertex::time() 997
```

Finally, we switch to the last sample stream and invoke the (s, t) strategy via a call to ConnectBDPT(). The final radiance estimate is multiplied by the inverse probability of choosing the current strategy (s, t) , which is equal to nStrategies.

```
(Execute connection strategy and return the radiance estimate) ≡ 1036
    sampler.StartStream(connectionStreamIndex);
    return ConnectBDPT(scene, lightVertices, cameraVertices, s, t,
                        *lightDistr, *camera, sampler, pRaster) * nStrategies;
```

Main Rendering Loop

There are two phases to the rendering process implemented in MLTIntegrator::Render(). The first phase generates a set of bootstrap samples that are candidates for initial states of Markov chains and computes the normalization constant $b = \int I(X) d\Omega$, from Equation (16.22). The second phase runs a series of Markov chains, where each chain chooses one of the bootstrap samples for its initial sample vector and then applies Metropolis sampling.

```
(MLT Method Definitions) +≡
void MLTIntegrator::Render(const Scene &scene) {
    std::unique_ptr<Distribution1D> lightDistr =
        ComputeLightPowerDistribution(scene);
<Generate bootstrap samples and compute normalization constant b 1038>
<Run nChains Markov chains in parallel 1039>
<Store final image computed with MLT 1042>
}
```

Following the approach described in Section 13.4.3, we avoid issues with start-up bias by computing a set of bootstrap samples with a standard Monte Carlo estimator and using them to create a distribution that supplies the initial states of the Markov chains. This process builds on the sample generation and evaluation routines implemented previously.

Each bootstrap sample is technically a sequence of `maxDepth + 1` samples with different path depths; the following fragment initializes the array `bootstrapWeights` with their corresponding luminance values. At the end, we create a `Distribution1D` instance to sample bootstrap paths proportional to their luminances and set the constant `b` to the sum of average luminances for each depth. Because we've kept the contributions of different path sample depths distinct, we can preferentially sample path lengths that make the largest contribution to the image. It is straightforward to compute the bootstrap samples in parallel, as all loop iterations are independent from one another.

```
(Generate bootstrap samples and compute normalization constant b) ≡ 1038
int nBootstrapSamples = nBootstrap * (maxDepth + 1);
std::vector<Float> bootstrapWeights(nBootstrapSamples, 0);
std::vector<MemoryArena> bootstrapThreadArenas(MaxThreadIndex());
ParallelFor(
    [&](int i) {
        <Generate ith bootstrap sample 1039>
    }, nBootstrap, 4096);
Distribution1D bootstrap(&bootstrapWeights[0], nBootstrapSamples);
Float b = bootstrap.funcInt * (maxDepth + 1);
```

As usual, `maxDepth` denotes the maximum number of interreflections that should be considered. `nBootstrap` sets the number of bootstrapping samples to use to seed the iterations and compute the integral of the scalar contribution function, Equation (16.22).

```
(MLTIntegrator Private Data) +≡ 1035
const int maxDepth;
const int nBootstrap;
```

In each iteration, we instantiate a dedicated `MLTSampler` with index `rngIndex` providing a unique sample vector that is uniformly distributed in the primary sample space. Next, we evaluate L to obtain a corresponding radiance estimate for the current path depth `depth` and write its luminance into `bootstrapWeights`. The raster-space position `pRaster` is not needed in the preprocessing phase, so it is ignored.

```
ComputeLightPowerDistribution() 974
Distribution1D 758
Distribution1D::funcInt 758
Float 1062
MaxThreadIndex() 1093
MemoryArena 1074
MLTIntegrator::maxDepth 1038
MLTIntegrator::nBootstrap 1038
MLTSampler 1029
ParallelFor() 1088
Scene 23
```

```
(Generate ith bootstrap sample) ≡ 1038
    MemoryArena &arena = bootstrapThreadArenas[ThreadIndex];
    for (int depth = 0; depth <= maxDepth; ++depth) {
        int rngIndex = i * (maxDepth + 1) + depth;
        MLTSampler sampler(mutationsPerPixel, rngIndex, sigma,
                            largeStepProbability, nSampleStreams);
        Point2f pRaster;
        bootstrapWeights[rngIndex] =
            L(scene, arena, lightDistr, sampler, depth, &pRaster).y();
        arena.Reset();
    }
```

The `mutationsPerPixel` parameter is analogous to `Sampler::samplesPerPixel` and denotes the number of iterations that MLT (on average!) spends in each pixel. Individual 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 total number of Metropolis samples taken is the product of the number of pixels and `mutationsPerPixel`.

The `sigma` and `largeStepProbability` member variables give the corresponding configuration parameters of the `MLTSampler`.

```
(MLTIntegrator Private Data) +≡ 1035
    const int mutationsPerPixel;
    const Float sigma, largeStepProbability;
```

We now move on to the main rendering task, which performs a total of `nTotalMutations` mutation steps spread out over `nChains` parallel Markov chains.

We must be careful that the actual number of mutation steps indeed comes out to be equal to `nTotalMutations`, particularly when `nTotalMutations` is not divisible by the number of parallel chains. The solution is simple: we potentially stop the last Markov chain a few iterations short; the corrected number of per-chain iterations is given by `nChainMutations`.

Film 484
`Film::GetSampleBounds()` 487
 Float 1062
`MemoryArena` 1074
`MemoryArena::Reset()` 1076
`MLTIntegrator::L()` 1036
`MLTIntegrator::maxDepth` 1038
`MLTIntegrator:::`
 `mutationsPerPixel`
 1039
`MLTIntegrator::nChains` 1040
`MLTSampler` 1029
`ParallelFor()` 1088
 Point2f 68
`Sampler::samplesPerPixel` 422
`Spectrum::y()` 325
`ThreadId` 1089

```
(Run nChains Markov chains in parallel) ≡ 1038
    Film &film = *camera->film;
    int64_t nTotalMutations = (int64_t)mutationsPerPixel *
        (int64_t)film.GetSampleBounds().Area();
    ParallelFor(
        [&](int i) {
            int64_t nChainMutations =
                std::min((i + 1) * nTotalMutations / nChains,
                         nTotalMutations) - i * nTotalMutations / nChains;
            (Follow ith Markov chain for nChainMutations 1040)
        }, nChains);
```

`nChains` specifies the number of Markov chains that should be executed independently of each other. Its default value of 100 is a trade-off between providing sufficient parallelism and running each chain for a long amount of time.

(MLTIntegrator Private Data) +≡

```
const int nChains;
```

1035

The MLT integrator only splats contributions to arbitrary pixels in the film; it doesn't fill in well-defined tiles of the image plane. Therefore, a `FilmTile` isn't necessary, and calls to `Film::AddSplat()` suffice to update the image.

(Follow ith Markov chain for nChainMutations) ≡

1039

```
MemoryArena arena;
```

(Select initial state from the set of bootstrap samples 1040)

(Initialize local variables for selected state 1040)

(Run the Markov chain for nChainMutations steps 1041)

Every Markov chain instantiates its own pseudo-random number generator following a unique stream. Note that this RNG instance is separate from the one in `MLTSampler`: it is used to pick the initial state and to accept or reject Metropolis proposals later on. Due to the ordering used earlier when initializing the entries of bootstrap array, we can immediately deduce the path depth of the sampled index `bootstrapIndex`. An important consequence of the method used to generate the bootstrap distribution is that the expected number of sampled initial states with a given depth value is proportional to their contribution to the image.

(Select initial state from the set of bootstrap samples) ≡

1040

```
RNG rng(i);
int bootstrapIndex = bootstrap.SampleDiscrete(rng.UniformFloat());
int depth = bootstrapIndex % (maxDepth + 1);
```

Having chosen the bootstrap sample, we must now obtain the corresponding primary sample space vector \mathbf{X} . One way to accomplish this would have been to store all `MLTSampler` instances in the preprocessing phase. A more efficient approach builds on the property that the initialization in the `MLTSampler` constructor is completely deterministic and just depends on the `rngSequenceIndex` parameter. Thus, here we can create an `MLTSampler` with index `bootstrapIndex`, which recreates the exact same sampler that originally produced the sampled entry of the bootstrap distribution.

With the sampler in hand, we can compute the current value of L and its position on the film.

(Initialize local variables for selected state) ≡

1040

```
MLTSampler sampler(mutationsPerPixel, bootstrapIndex, sigma,
                    largeStepProbability, nSampleStreams);
Point2f pCurrent;
Spectrum LCurrent =
    L(scene, arena, lightDistr, sampler, depth, &pCurrent);
```

The implementation of the Metropolis sampling routine in the following fragments follows 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, and the weighted contributions of both the new and old samples are recorded. The proposed mutation is then randomly accepted based on its acceptance probability.

```
DistributionID::
    SampleDiscrete()
760

Film::AddSplat() 494
FilmTile 489
MemoryArena 1074
MLTIntegrator::L() 1036
MLTIntegrator::
    largeStepProbability
1039
MLTIntegrator::maxDepth 1038
MLTIntegrator::
    mutationsPerPixel
1039
MLTIntegrator::nSampleStreams
1036
MLTIntegrator::sigma 1039
MLTSampler 1029
Point2f 68
RNG 1065
RNG::UniformFloat() 1066
Spectrum 315
```

One of the unique characteristics of MMLT (Section 16.4.2) compared to other Metropolis-type methods is that each Markov chain is restricted to paths of a fixed depth value. The first sample dimension selects among various different strategies, but only those producing equal path depths are considered, which improves performance of the method by making proposals more local. The contribution of all path depths is accounted for by starting many Markov chains with different initial states.

```
<Run the Markov chain for nChainMutations steps> ≡ 1040
    for (int64_t j = 0; j < nChainMutations; ++j) {
        sampler.StartIteration();
        Point2f pProposed;
        Spectrum LProposed =
            L(scene, arena, lightDistr, sampler, depth, &pProposed);
<Compute acceptance probability for proposed sample 1041>
<Split both current and proposed samples to film 1041>
<Accept or reject the proposal 1042>
    }
```

Given the scalar contribution function's value, the acceptance probability is then given by the simplified expression from Equation (13.8) thanks to the symmetry of our mutations on primary sample space.

As described at the start of the section, the spectral radiance value $L(X)$ 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. Here, we compute the path's luminance, which is a reasonable choice.

```
<Compute acceptance probability for proposed sample> ≡ 1041
    Float accept = std::min((Float)1, LProposed.y() / LCurrent.y());
```

Both samples can now be added to the image. Here, they are scaled with weights based on the expected values optimization introduced in Section 13.4.1.¹²

```
<Split both current and proposed samples to film> ≡ 1041
    if (accept > 0)
        film.AddSplat(pProposed, LProposed * accept / LProposed.y());
        film.AddSplat(pCurrent, LCurrent * (1 - accept) / LCurrent.y());
Film::AddSplat() 494
Float 1062
MLTIntegrator::L() 1036
MLTSampler 1029
MLTSampler::StartIteration() 1031
Point2f 68
PrimarySample 1030
Spectrum 315
Spectrum::y() 325
```

Finally, the proposed mutation is either accepted or rejected, based on the computed acceptance probability `accept`. If the mutation is accepted, then the values `pProposed` and `LProposed` become properties of the current state. In either case, the `MLTSampler` must be informed of the outcome so that it can update the `PrimarySample` accordingly.

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 would halve the number of `Film::AddSplat()` calls while still computing the same result.

(Accept or reject the proposal) \equiv

```

if (rng.UniformFloat() < accept) {
    pCurrent = pProposed;
    LCurrent = LProposed;
    sampler.Accept();
} else
    sampler.Reject();

```

1041

Metropolis sampling only considers the *relative* frequency of samples and cannot create an image that is correctly scaled in *absolute* terms; hence the value b is crucial: it contains an estimate of the average luminance of the Film that we use to remove this ambiguity. Each Metropolis iteration within *(Run nChains Markov chains in parallel)* has splatted contributions with weighted unit luminance to the Film so that the final average film luminance before Film::WriteImage() is exactly equal to mutationsPerPixel. We thus must cancel this factor out and multiply by b when writing the image to convert to actual incident radiance on the film.

(Store final image computed with MLT) \equiv

1038

```
camera->film->WriteImage(b / mutationsPerPixel);
```

FURTHER READING

The general idea of tracing light-carrying paths from 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. Christensen (2003) surveyed applications of adjoint functions and importance to solving the LTE and related problems.

Sources of non-symmetric scattering and their impact on bidirectional light transport algorithms were first identified by Veach (1996).

Pharr and Humphreys (2004) proposed the method to sample emitted rays from environment map light sources that is used in this chapter. Dammertz and Hanika (2009) described a variation on this approach that sampled points on the visible faces of the scene bounding box rather than an oriented disk; this can lead to fewer wasted samples.

Photon Mapping

Approaches like Arvo's caustic rendering algorithm (Arvo 1986) formed the basis for an improved technique that stored illumination in texture maps on surfaces developed by Collins (1994). Density estimation techniques for global illumination were first introduced by Shirley, Walter, and collaborators (Shirley et al. 1995; Walter et al. 1997).

Jensen (1995, 1996) developed the photon mapping algorithm, which introduced the key innovation of storing the light contributions in a general 3D data structure rather than in texture maps. Important improvements to the photon mapping method are described in follow-up papers and a book by Jensen (1996, 1997, 2001).

Final gathering for finite-element radiosity algorithms was first described in Reichert's thesis (Reichert 1992). If the full photon map is stored in memory, the directional dis-

```

Camera::film 356
Film 484
Film::WriteImage() 494
MLTIntegrator::camera 1037
MLTIntegrator::
    mutationsPerPixel
    1039
MLTSampler::Accept() 1031
MLTSampler::Reject() 1034
RNG::UniformFloat() 1066

```

tribution of photons can be used to construct optimized final gathering techniques that importance sample directions that are likely to have large contributions (Jensen 1995). 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. In another paper, Spencer and Jones (2009b) showed that a simple iterative relaxation scheme to reduce clumping in photon maps can lead to dramatic improvements in the quality of density estimates.

Havran et al. (2005) developed a final gathering photon mapping 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. 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. Hachisuka et al. (2008b) developed the progressive photon mapping algorithm; stochastic progressive photon mapping was developed by Hachisuka and Jensen (2009).

The advantages of SPPM over traditional photon mapping are significant, and the approach was quickly adopted after its introduction. Hachisuka et al. (2010) showed how to use arbitrary density estimation kernels and how to compute error estimates during rendering to automatically determine when to stop further iterations. Knaus and Zwicker (2011) re-derived SPPM following a different approach and showed that it was possible to only maintain global statistics for values like the current search radius rather than having a separate value for each pixel. See Kaplanyan and Dachsbacher (2013a) for an extensive study of SPPM's convergence rates and an improved (but more complex) method for updating SPPM estimates after each iteration.

The question of how to find the most effective set of photons for photon mapping is an important one: 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). The earliest applications of Metropolis sampling to photon mapping was proposed in Wald's Diploma thesis (1999). Fan et al. (2005) showed that the application of Veach's particle-tracing theory to photon mapping provides a mechanism for generating photon paths starting from the camera. They were able to use this approach in conjunction with a Metropolis sampling algorithm to generate photon distributions. Hachisuka and Jensen (2011) used Metropolis sampling to find photon paths that were visible to the camera; their algorithm is notable for both its effectiveness and its ease of implementation. Chen et al. (2011) use a similar approach but sample additional terms of the path contribution function and distribute additional photons to parts of the image with higher error.

Jensen and Christensen (1998) were the first to generalize the photon mapping algorithm to participating media. Knaus and Zwicker (2011) showed how to render participating media using SPPM. Jarosz et al. (2008) had the important insight that expressing the scattering integral over a beam through the medium as the measurement to be evaluated could make photon mapping's rate of convergence much higher than if a series of point photon estimates was instead taken along each ray. Section 5.6 of Hachisuka's thesis

(2011) and Jarosz et al. (2011a, 2011b) showed how to apply this approach progressively. For another representation, see Jakob et al. (2011), who fit a sum of anisotropic Gaussians to the equilibrium radiance distribution in participating media.

Bidirectional Path Tracing

Bidirectional path tracing was independently developed by Lafortune and Willem (1994) and Veach and Guibas (1994). The development of multiple importance sampling was integral to the effectiveness of bidirectional path tracing (Veach and Guibas 1995). Lafortune and Willem (1996) showed how to apply bidirectional path tracing to rendering participating media, and Kollig and Keller (2000) showed how bidirectional path tracing can be modified to work with quasi-random sample patterns.

An exciting recent development has been simultaneous work by Hachisuka et al. (2012) and Georgiev et al. (2012), who developed a unified framework for both photon mapping and bidirectional path tracing. Their approaches allowed photon mapping to be included in the path space formulation of the light transport equation, which in turn made it possible to derive light transport algorithms that use both approaches to generate paths and combine them using multiple importance sampling.

Kaplanyan and Dachsbacher (2013b) noted that photon mapping algorithms use illumination from nearby points even in cases where unbiased approaches are effective. They developed a technique for regularization of light-carrying paths, where an unbiased path tracer or bidirectional path tracer is modified to treat delta distributions that cause impossible-to-sample configurations instead as having non-zero value over a small cone of directions. Thus, bias is introduced only in the challenging settings.

Vorba et al. (2014) developed an approach to compute effective sampling distributions for difficult lighting configurations over the course of rendering rather than in a preprocess and showed its applicability to bidirectional path tracing.

Metropolis Light Transport

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). Pauly, Kollig, and Keller (2000) generalized the MLT algorithm to include volume scattering. Pauly's thesis (Pauly 1999) described the theory and implementation of bidirectional and Metropolis-based algorithms for volume light transport.

Fan et al. (2005) developed a method that let the user explicitly specify a number of important paths (e.g., through a tricky geometric configuration) that could then be used as a target state in Metropolis mutations. The energy redistribution path tracing algorithm by Cline et al. (2005) starts one or more Markov chains at every pixel of the image and runs them for a small number of iterations; the method is notable for being unbiased despite its use of non-ergodic Markov chains that can only explore a subset of path space.

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).

Kelemen et al. (2002) developed the “primary sample space MLT” formulation of Metropolis light transport. They also suggested the approach implemented in the `MLTSampler` for lazily updating sample vector components when performing mutations. Hachisuka et al. (2014) developed the MMLT approach that is implemented in the `MLTIntegrator` in this chapter.

The optimal choice of the large step probability is scene dependent: for scenes with difficult-to-sample transport paths, it’s better for it to be lower, so that more successful mutations are performed with small steps once a good path is found. For scenes with simpler light transport, it’s better for the probability to be higher, so that the overall path space is explored more thoroughly. Zsolnai and Szirmay-Kalos (2013) developed a technique that gathered statistics about paths during the bootstrap phase that made it possible to automatically set this parameter to a near-optimal value.

Other Rendering Approaches

A number of algorithms have been developed based on a first phase of computation that traces paths from the light sources to create “virtual lights,” where these lights are then used to approximate indirect illumination during a second phase. The principles behind this approach were first introduced by Keller’s work on *instant radiosity* (1997). The more general instant global illumination algorithm was developed by Wald, Benthin, and collaborators (Wald et al. 2002, 2003; Benthin et al. 2003). See Dachsbacher et al.’s recent survey article (2014) for a summary of recent work in this area.

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.

Bidirectional lightcuts (Walter et al. 2012) trace longer subpaths from the camera to obtain a family of light connection strategies; combining the strategies using multiple importance sampling eliminates bias artifacts that are commonly produced by virtual point light methods.

Jakob and Marschner (2012) expressed light transport involving specular materials as an integral over a high-dimensional manifold embedded in path space. A single light path corresponds to a point on the manifold, and nearby paths are found using a local parameterization that resembles Newton’s method; they applied a Metropolis-type method through this parameterization to explore the neighborhood of challenging specular and near-specular configurations.

Hanika et al. (2015a) apply an improved version of the local path parameterization in a pure Monte Carlo context to estimate the direct illumination through one or more dielectric boundaries; this leads to significantly better convergence when rendering glass-enclosed objects or contaminated surfaces with water droplets.

Kaplanyan et al. (2014) observed that the path contribution function is close to being separable when paths are parameterized using the endpoints and the half-direction vectors at intermediate vertices, which are equal to the microfacet normals in the context of microfacet reflectance models. Performing Metropolis sampling in this half-vector domain leads to a method that is particularly good at rendering glossy interreflection. An extension by Hanika et al. (2015b) improves the robustness of this approach and proposes an optimized scheme to select mutation sizes to reduce sample clumping in image space.

Another interesting approach was developed by Lehtinen and collaborators (Lehtinen et al. 2013, Manzi et al. 2014). Building on the observation that ideally, most samples from the path space should be taken around discontinuities (and not in smooth regions of the image), they developed a measurement contribution function for Metropolis sampling that focused samples on gradients in the image. They then reconstructed high-quality final images from horizontal and vertical gradient images and a coarse, noisy image. More recently, Kettunen et al. (2015) showed how this approach could be applied to regular path tracing, without Metropolis sampling. Manzi et al. (2015) showed its application to bidirectional path tracing.

Hair is particularly challenging to render; not only is it extremely geometrically complex but multiple scattering among hair also 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.

While the rendering problem as discussed so far has been challenging enough, Jarabo et al. (2014a) showed the extension of the path integral to not include the steady-state assumption—i.e., accounting for the non-infinite speed of light. Time ends up being extremely high frequency, which makes rendering challenging; they showed successful application of density estimation to this problem.

EXERCISES

- ② 16.1 Derive importance functions and implement the Camera `We()`, `Pdf_We()`, `Sample_Wi()`, and `Pdf_Wi()` methods for one or more of `EnvironmentCamera`, `OrthographicCamera`, or `RealisticCamera`. Render images using bidirectional path tracing or MMLT and show that given sufficient samples, they converge to the same images as when the scene is rendered with standard path tracing.
- ② 16.2 Discuss situations where the current methods for sampling outgoing rays from `ProjectionLights` and `GonioPhotometricLights` may be extremely inefficient, choosing many rays in directions where the light source casts no illumination. Use the `Distribution2D` structure to implement improved sampling techniques for each of them based on sampling from a distribution based on the luminance in their 2D image maps, properly accounting for the transformation from the 2D image map sampling distribution to the distribution of directions on the sphere. Verify that the system still computes the same images (modulo variance) with your new sampling techniques when using an `Integrator` that calls

`Distribution2D` 785
`EnvironmentCamera` 376
`GonioPhotometricLight` 728
`Integrator` 25
`OrthographicCamera` 361
`ProjectionLight` 724
`RealisticCamera` 378

these methods. Determine how much efficiency is improved by using these sampling methods instead of the default ones.

- ③ 16.3 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 lightcuts? 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 bidirectional integrators.
- ① 16.4 Experiment with the parameters to the SPPMIntegrator until you get a good feel for how they affect rendering time and the appearance of the final image. At a minimum, experiment with varying the search radius, the number of photons traced, and the number of iterations.
- ② 16.5 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 weight and end up being used by visible points. 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 additional bias into the shooting algorithm.) Derive and implement such an approach. Show that in the limit, your modified SPPMIntegrator computes the same results as the original. How much do these changes improve the rate of convergence?
- ② 16.6 The SPPMIntegrator ends up storing all of the BSDFs along camera paths to the first visible point, even though only the last BSDF is needed. First, measure how much memory is used to store unnecessary BSDFs in the current implementation for a variety of scenes. Next, modify the VisiblePoint representation to store the reflectance, BSDF::rho(), at visible points and then compute reflection assuming a Lambertian BSDF. Does this approximation introduce visible error in test scenes? How much memory is saved?
- ② 16.7 To find the VisiblePoints around a photon–surface intersection, the SPPM Integrator uses a uniform grid to store the bounding boxes of visible points expanded by their radii. Investigate other spatial data structures for storing visible points that support efficient photon/nearby visible point queries, and implement an alternate approach. (You may want to specifically consider octrees and kd-trees.) How do performance and memory use compare to the current implementation?
- ② 16.8 Implement “final gathering” in the SPPMIntegrator, where camera rays are followed for one more bounce after hitting a diffuse surface. Investigate how many iterations and how many photons per iteration are needed to get good results with this approach for a variety of scenes compared to the current implementation.
- ② 16.9 There is actually one case where collisions from the hash() function used by the SPPMIntegrator can cause a problem: if, for example, nearby voxels have

BSDF::rho() 575

hash() 982

SPPMIntegrator 973

VisiblePoint 979

a collision, then a `VisiblePoint` that overlaps both of them will be placed in a linked list twice, and then a photon that is close to them will incorrectly contribute to the pixel's value twice. Can you prove that this will never happen for the current hash function? If it does happen, can you construct a scene where the error makes a meaningful difference in the final image? How might this problem be addressed?

- ➊ 16.10 Extend the SPPM integrator to support volumetric scattering. First, read the papers by Knaus and Zwicker (2011) and Jarosz et al. (2011b) and choose one of these approaches. Compare the efficiency and accuracy of images rendered with your implementation to rendering using the bidirectional path tracer or the MMLT integrator.
- ➋ 16.11 One shortcoming of the current `SPPMIntegrator` implementation is that it's inefficient for scenes where the camera is only viewing a small part of the overall scene: many photons may need to be traced to find ones that are visible to the camera. Read the paper by Hachisuka and Jensen (2011) on using adaptive Markov chain sampling to generate photon paths and implement their approach. Construct a scene where the current implementation is inefficient and your new one does much better, and render comparison images using equal amounts of computation for each. Are there any scenes where your implementation computes worse results?
- ➌ 16.12 Extend the BDPT integrator to support subsurface scattering with BSSRDFs. In addition to connecting pairs of vertices by evaluating the geometric term and BSDFs, your modified integrator should also evaluate the BSSRDF $S(p_o, \omega_o, p_i, \omega_i)$ when two points are located on an object with the same `Material` with a non-nullptr-valued BSSRDF. Since two connection techniques lead to paths with a fundamentally different configuration—straight-line transport versus an additional subsurface scattering interaction on the way from p_i on p_o —their area product density should never be compared to each other when computing multiple importance sampling weights.
- ➍ 16.13 Implement Russian roulette to randomly skip tracing visibility rays for low-contribution connections between subpaths in the `ConnectBDPT()` function. Measure the change in Monte Carlo efficiency compared to the current `BDPTIntegrator` implementation.
- ➎ 16.14 Modify the BDPT integrator to use the path space regularization technique described by Kaplanyan and Dachsbacher (2013b). (Their method makes it possible for light transport algorithms based on incremental path construction to still handle difficult sampling cases based on chains of specular interactions.) Construct a scene where this approach is useful, and render images to compare results between this approach, SPPM, and an unmodified bidirectional path tracer.
- ➏ 16.15 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 `MLTIntegrator`. For example, if only the `PrimarySample` values for the light subpath are mutated, then the camera subpath can be

`BDPTIntegrator` 992

`BSSRDF` 692

`ConnectBDPT()` 1009

`MLTIntegrator` 1035

`PrimarySample` 1030

`SPPMIntegrator` 973

`VisiblePoint` 979

reused. (An analogous case holds for the light subpath.) Even if a mutation is proposed for a subpath, if the samples for its first few vertices are left unchanged, then that portion of the path doesn't need to be retraced.

Modify the `MLTIntegrator` to add one or more of the above sampling strategies, and update 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 among samples. Is the Monte Carlo efficiency of your modified version better than the original implementation?

- ③ **16.16** 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.



CHAPTER SEVENTEEN

17 RETROSPECTIVE AND THE FUTURE

`pbrt` represents one single point in the space of rendering system designs. The basic decisions we made early on—that ray tracing would be the geometric visibility algorithm used, that physical correctness would be a cornerstone of the system, and that Monte Carlo would be the main approach used for numerical integration—all had pervasive implications for the system’s design. For example, an entirely different set of trade-offs would have been made if `pbrt` were a renderer designed instead for real-time performance and only rendering scenes with direct illumination.

This chapter first looks back at some of the details of the complete system, discusses some design alternatives, and also covers some potential major extensions that are more complex than have been described in the exercises. We then discuss recent trends in graphics hardware architectures and their implications for rendering systems like `pbrt`.

17.1 DESIGN RETROSPECTIVE

One of the basic assumptions in `pbrt`’s design was that the most interesting types of images to render are images with complex geometry and lighting and that supporting a wide variety of shapes, materials, light sources, and light transport algorithms was important. We also assumed that rendering these images well—with good sampling patterns, ray differentials, and antialiased textures—is worth the computational expense. One result of these assumptions is that `pbrt` is relatively inefficient at rendering simple scenes, where a more specialized system could do much better.

For example, a performance implication of our design priorities is that finding the BSDF at a ray intersection is more computationally expensive than it is in renderers that don’t expend as much effort filtering textures and computing ray differentials. We believe that

this effort pays off overall by reducing the need to trace more camera rays to address texture aliasing, although, again, for simple scenes, texture aliasing is often not a problem. On the other hand, most of the integrators in `pbrt` assume that hundreds or even thousands of samples will be taken in each pixel for high-quality global illumination; the benefits of high quality filtering are reduced in this case, since the high pixel sampling rate ends up sampling textures at a high rate as well.

The simplicity of some of the interfaces in the system can lead to unnecessary work being done. For example, the `Sampler` always computes lens and time samples, even if they aren't needed by the `Camera`; there's no way for the `Camera` to communicate its sampling needs. Similarly, if an `Integrator` doesn't use all of the array samples from its earlier calls to `Request1DArray()` and `Request2DArray()` for some ray, then the `Sampler`'s work for generating those samples is wasted. (This case can occur, for example, if the ray doesn't intersect any geometry.) For cases like these, we believe that the benefits to readers of making the system easier to understand outweigh the relatively small efficiency losses.

Throughout the book, we have tried to always add an exercise at the end of the chapter when we've known that there was an important design alternative or where we made an implementation trade-off that would likely be made differently in a production rendering system. (For example, Exercise 7.2 discusses the first issue with `Samplers` in the previous paragraph.) It's worth reading the exercises even if you don't plan to do them.

17.1.1 TRIANGLES ONLY

Another instance where the chosen abstractions in `pbrt` impact the overall system efficiency is the range of geometric primitives that the renderer supports. While ray tracing's ability to handle a wide variety of shapes is elegant, this property is not as useful in practice as one might initially expect. Most real-world scenes are either modeled directly with polygons or with smooth surfaces like spline patches and subdivision surfaces that either have difficult-to-implement or relatively inefficient ray–shape intersection algorithms. As such, they are usually tessellated into triangles for ray intersection tests in practice. Not many shapes that are commonly encountered in real-world scenes can be represented accurately with spheres and cones!

There are some advantages to designing a ray tracer around a single low-level shape representation like triangles and only operating on this representation throughout much of the pipeline. Such a renderer could still support a variety of primitives in the scene description but would always tessellate them at some point before performing intersection tests. Advantages of this design include:

- The renderer can depend on the fact that the triangle vertices can be transformed into world or camera space in advance, so no transformations of rays into object space are necessary (except when object instancing is used).
- The acceleration structures can be specialized so that their nodes directly store the triangles that overlap them. This improves the locality of the geometry in memory and enables ray–primitive intersection tests to be performed directly in the traversal routine, without needing to pass through two levels of virtual function calls to do so, as is currently the case in `pbrt`.

`Camera` 356

`Integrator` 25

`Sampler` 421

`Sampler::Request1DArray()`
423

`Sampler::Request2DArray()`
423

- Displacement mapping, where geometry is subdivided into small triangles, which can then have their vertices perturbed procedurally or with texture maps, can be more easily implemented if all primitives are able to tessellate themselves.

These advantages are substantial, for both increased performance and the complexity that they remove from many parts of the system. For a production renderer, rather than one with pedagogical goals like pbrt, this alternative is worth considering carefully. (Alternatively, triangles alone could be given special treatment—stored directly in acceleration structures and so forth—while other shapes were handled with a less efficient general purpose code path.)

17.1.2 INCREASED SCENE COMPLEXITY

Given well-built acceleration structures, a strength of ray tracing is that the time spent on ray-primitive intersections grows slowly with added scene complexity. As such, the maximum complexity that a ray tracer can handle may be limited more by memory than by computation. Because rays may pass through many different regions of the scene during a short period of time, virtual memory often performs poorly when ray tracing complex scenes due to the resulting incoherent memory access patterns.

One way to increase the potential complexity that a renderer is capable of handling is to reduce the memory used to store the scene. For example, pbrt currently uses approximately 4 GB of memory for the 24 million triangles in the landscape scene on the cover and in Figure 4.1. This works out to an average of 167 bytes per triangle. We have previously written ray tracers that managed an average of 40 bytes per triangle for scenes like these—at least a 4× reduction is possible.

Reducing memory overhead requires careful attention to memory use throughout the system. For example, in the aforementioned system, we provided three different `Triangle` implementations, one using 8-bit `uint8_ts` to store vertex indices, one using 16-bit `uint16_ts`, and one using 32-bit `uint32_ts`. The smallest index size that was sufficient for the range of vertex indices in the mesh was chosen at run time. Deering’s paper on geometry compression (Deering 1995) and Ward’s packed color format (Ward 1992) are both good inspirations for thinking along these lines. See the “Further Reading” section in Chapter 4 for information about more memory-efficient acceleration structure representations.

A more complex approach to implement is geometry caching (Pharr and Hanrahan 1996), where the renderer holds a fixed amount of geometry in memory and discards geometry that hasn’t been accessed recently. This approach is useful for scenes with a lot of tessellated geometry, where a compact higher level shape representation like a subdivision surface can explode into a large number of triangles. When available memory is low, some of this geometry can be discarded and regenerated later if needed. Geometry stored on disk can also be loaded into geometry caches; with the advent of economical flash storage offering hundreds of megabytes per second of read bandwidth, this approach is even more attractive.

The performance of such a cache can be substantially improved by reordering the rays that are traced in order to improve their spatial and thus memory coherence (Pharr et

al. 1997). An easier-to-implement and more effective approach to improving the cache’s behavior was described by Christensen et al. (2003), who wrote a ray tracer that uses simplified representations of the scene geometry in a geometry cache. More recently, Yoon et al. (2006), Budge et al. (2009), Moon et al. (2010), and Hanika et al. (2010) have developed improved approaches to this problem. See Rushmeier, Patterson, and Veerasamy (1993) for an early example of how to use simplified scene representations when computing indirect illumination.

17.1.3 PRODUCTION RENDERING

Rendering high-quality imagery for film introduces a host of challenges beyond the topics discussed in this book. Being able to render highly complex scenes—with both geometric and texture complexity—is a requirement. Most production renderers have deferred loading and caching of texture and geometry at the hearts of their implementations. Programmable surface shaders are also critical for allowing users to specify complex material appearances.

Another practical challenge is integrating with interactive modeling and shading tools: it’s important that artists be able to quickly see the effect of changes that they make to models, surfaces, and lights. Deep integration with tools is necessary for this to work well—communicating the scene description from scratch with a text file each time the scene is rendered, as is done in `pbrt`, is not a viable approach.

Unfortunately, the developers of most of the current crop of production rendering systems haven’t yet followed the lead of Cook et al. (1987), who described Reyes and its design in great detail. Exceptions include PantaRay, which was used by Weta Digital and is described by Pantaleoni et al. (2010), and Disney’s Hyperion renderer (Eisenacher et al. 2013).

17.1.4 SPECIALIZED COMPILATION

The OptiX ray-tracing system, which is described by Parker et al. (2010), has a very interesting system structure: it’s a combination of built-in functionality (e.g., for building acceleration structures and traversing rays through them) that can be extended by user-supplied code (for primitive implementations, surface shading functions, etc.). Many renderers over the years have allowed user extensibility of this sort, usually through some kind of plug-in architecture. OptiX is distinctive in that it is built using a run-time compilation system that compiles all of this code together.

Because the compiler has a view of the entire system when generating code, the resulting custom renderer can be automatically specialized in a variety of ways. For example, if the surface shading code never uses the (u, v) texture coordinates, the code that computes them in the triangle shape intersection test can be optimized out as dead code. Or, if the ray’s time field is never accessed, both the code that sets it and even the structure member itself can be eliminated. Thus, this approach allows a degree of specialization (and resulting performance) that would be difficult to achieve manually, at least for more than a single system variant.

17.2 ALTERNATIVE HARDWARE ARCHITECTURES

Our focus in this book has been on traditional multi-core CPUs as a target for the system. Furthermore, we have ignored the potential of being able to perform up to eight floating-point operations per instruction by using CPU SIMD hardware. While other computing architectures like GPUs or specialized ray-tracing hardware are appealing targets for a renderer, their characteristics tend to change rapidly, and their programming languages and models are less widely known than languages like C++ on CPUs. Though we haven't targeted these architectures with *pbrt*, it's useful to discuss their characteristics.

The early days of ray tracing saw work in this area focused on multiprocessors (Cleary et al. 1983; Green and Paddon 1989; Badouel and Priol 1989) and clusters of computers (Parker et al. 1999; Wald et al. 2001a, 2002; Wald, Slusallek, and Benthin 2001; Wald, Benthin, and Slusallek 2003). More recently, substantial capabilities have become available in single computer systems, which has led to a shift of focus to the capabilities of CPU SIMD units and GPUs.

CPUs have long been designed to run a single thread of computation as efficiently as possible; these processors can be considered to be latency focused, in that the goal is to finish a single computation as quickly as possible. (Only since 2005 or so has this focus started to slowly change in CPU design, as multicore CPUs have provided a small number of independent latency-focused processors on a single chip.) Starting with the advent of programmable graphics processors around the year 2003, *throughput processors* (as exemplified by GPUs) have increasingly become the source of most of the computational capability available in many computer systems. These processors focus not on single-thread performance but instead on efficiently running hundreds or thousands of computations in parallel with high aggregate computational throughput, without trying to minimize time for any of the individual computations.

By not focusing on single-thread performance, throughput processors are able to devote much less space on the chip for caches, branch prediction hardware, out-of-order execution units, and other features that have been invented to improve single-thread performance on CPUs. Thus, given a fixed amount of chip area, these processors are able to provide many more arithmetic logic units (ALUs) than a CPU. For the types of computations that can provide a substantial amount of independent parallel work, throughput processors can keep these ALUs busy and very efficiently execute the computation. As of the time of writing, GPUs offer approximately ten times as many peak FLOPS as high-end CPUs; this makes them highly attractive for many processing-intensive tasks (including ray tracing).¹

Single instruction, multiple data (SIMD) processing, where processing units execute a single instruction across multiple data elements, is the key mechanism that throughput processors use to efficiently deliver computation; both today's CPUs and today's GPUs

¹ However, graphics processors typically consume more power and are physically larger chips than CPUs; some of their improved performance comes purely from using more power and more chip area. A fair comparison is to consider performance per watt or per square millimeter of silicon, which puts GPUs at 3–5× more capable in terms of peak performance.

have SIMD vector units in their processing cores. Modern CPUs generally have a handful of processing cores and support four or eight 32-bit floating point operations in their vector instruction sets (e.g., SSE, NEON, or AVX). GPUs currently have tens of processing cores,² each with SIMD vector units between 8 and 64 elements wide. (Intel's Xeon Phi architecture, which features over 50 relatively simple CPU cores, each with a 16-wide 32-bit floating-point SIMD unit, lies somewhere between these two points.) It is likely that both the number of processing cores and the width of the vector units in all of these processor architectures will go up over time, as hardware designers make use of additional transistors made possible by Moore's law.

17.2.1 GPU RAY TRACING

Purcell et al. (2002, 2003) and Carr, Hall, and Hart (2002) were the first to map general-purpose ray tracers to throughput graphics processors. GPU-based construction of data structures tends to be challenging; see Zhou et al. (2008), Lauterbach et al. (2009), Pantaleoni and Luebke (2010), Garanzha et al. (2011), and Karras and Aila (2013) for techniques for building kd-trees and BVHs on GPUs.

Aila and Laine (2009) carefully investigated the performance of SIMD ray tracing on a graphics processor, using their insights to develop a new SIMD-friendly traversal algorithm that was substantially more efficient than the previous best known approach. Their insights are worth careful consideration by all implementors of high-performance rendering systems.

A big challenge in using throughput processors for rendering systems can be finding coherent collections of computation that use the SIMD vector elements efficiently. Consider a Monte Carlo path tracer tracing a collection of rays; after random sampling at the first bounce, each ray will in general intersect completely different objects, likely with completely different surface shaders. At this point, running the surface shaders will likely make poor use of SIMD hardware as each ray needs to execute a different computation. This specific problem of efficient shading was investigated by Hoberock et al. (2009), who resorted a large number of intersection points to create coherent collections of work before executing their surface shaders.

Another challenge is that relatively limited amount of local memory on GPUs makes it challenging to implement light transport algorithms that require more than a small amount of storage for each ray. (For example, even storing all of the vertices of a pair of subpaths for a bidirectional path tracing algorithm is not straightforward.) The paper by Davidović et al. (2014) gives a thorough overview of these issues and previous work and includes a discussion of implementations of a number of sophisticated light transport algorithms on the GPU.

An interesting trade-off for renderer developers to consider is exhibited by Hachisuka's path tracer, which uses a rasterizer with parallel projection to trace rays, effectively computing visibility in the same direction for all of the points being shaded (Hachisuka

² The definition of a "core" on a throughput processor is notoriously tricky, with different hardware vendors promoting different definitions. Here, we are following the relatively vendor-neutral terminology proposed by Fatahalian (2008).

2005). His insight was that although this approach doesn't give a particularly good sampling distribution for Monte Carlo path tracing, in that each point isn't able to perform importance sampling to select outgoing directions, the increased efficiency from computing visibility for a very coherent collection of rays paid off overall. In other words, for a fixed amount of computation, so many more samples could be taken using rasterization versus using ray tracing that the much larger number of less well-distributed samples generated a better image than a smaller number of well-chosen samples. We suspect that this general issue of trading off between computing exactly the locally desired result at a single point versus computing what can be computed very efficiently globally for many points will be an important one for developers to consider on the SIMD processors of the future.

17.2.2 PACKET TRACING

For narrow SIMD widths on CPUs (like four-element SSE), some performance gains can be attained by opportunistically using the SIMD unit. For example, one might modify `pbrt` to use SSE instructions for the operations defined in the `Spectrum` class, thus generally being able to do three floating-point operations per instruction (for RGB spectra) rather than just one if the SIMD unit was not used. This approach would achieve 75% utilization of an SSE unit for those instructions but doesn't help with performance in the rest of the system. In some cases, optimizing compilers can identify multiple computations in scalar code that can be executed together using a single SIMD instruction.

Achieving *excellent* utilization of SIMD vector units generally requires that the entire computation be expressed in a *data parallel* manner, where the same computation is performed on many data elements simultaneously. A natural way to extract data parallelism in a ray tracer is to have each processing core responsible for tracing n rays at a time, where n is at least the size of the SIMD width, if not larger; as such, each SIMD vector "lane" is responsible for just a single ray, and each vector instruction performs only a single scalar computation for each of the rays it's responsible for. Thus, high SIMD utilization comes naturally, except for the cases where some rays require different computations than others.

This approach has seen success with high-performance CPU ray tracers (where it is generally called "packet tracing"). Wald et al. (2001a) introduced this approach, which has since seen wide adoption. In a packet tracer, the camera generates "packets" of n rays that are then processed as a unit. Acceleration structure traversal algorithms are modified so that they visit a node if *any* of the rays in the packet passes through it; primitives in the leaves are tested for intersection with all of the rays in the packet, and so forth. Packet tracing has been shown to lead to substantial speedups, although it becomes increasingly less effective as the rays to be traced become less coherent; it works well for camera rays and shadow rays to localized light sources, since the packets of rays will pass through similar regions of the scene, but efficiency generally falls off with multi-bounce light transport algorithms. Finding ways to retain good efficiency with packet tracing remains an active area of research.

parameter rather than just a single ray, and so forth. In contrast, the vectorization in programs written for throughput processors like GPUs is generally implicit: code is written as if it just operates on a single ray at a time, but the underlying compiler and hardware actually execute one instance of the program in each SIMD lane.

For processors that directly expose their SIMD nature in their instruction sets (like CPUs or Intel's Xeon Phi), the designer of the programming model is able to choose whether to provide an implicit or an explicit vector model to the user. See Parker et al.'s (2007) ray-tracing shading language for an example of compiling an implicitly data-parallel language to a SIMD instruction set on CPUs. See also Georgiev and Slusallek's (2008) approach, where a generic programming approach is taken in C++ to allow implementing a high-performance ray tracer with details like packets well hidden. `ispc`, described in a paper by Pharr and Mark (2012), provides a general-purpose “single program multiple data” (SPMD) language for CPU vector units that also provides this model.

Reshetov et al. (2005) generalized packet tracing, showing that gathering up many rays from a single origin into a frustum and then using the frustum for acceleration structure traversal could lead to very high-performance ray tracing; they refined the frusta into subfrusta and eventually the individual rays as they reached lower levels of the tree. Reshetov (2007) later introduced a technique for efficiently intersecting a collection of rays against a collection of triangles in acceleration structure leaf nodes by generating a frustum around the rays and using it for first-pass culling. See Benthin and Wald (2009) for a technique to use ray frusta and packets for efficient shadow rays.

While packet tracing is effective for coherent collections of rays that follow generally the same path through acceleration structures, it's much less effective for incoherent collections of rays, which are more common with global illumination algorithms. To address this issue, Ernst and Greiner (2008), Wald et al. (2008), and Dammertz et al. (2008) proposed only traversing a single ray through the acceleration structure at once but improving SIMD efficiency by simultaneously testing each ray against a number of bounding boxes at each step in the hierarchy.

Another approach to the ray incoherence problem is to reorder small batches of incoherent rays to improve SIMD efficiency; representative work in this area includes papers by Mansson et al. (2007), Boulos et al. (2008), Gribble and Ramani (2008), and Tsakok (2009). More recently, Barringer and Akenine-Möller (2014) developed a SIMD ray traversal algorithm that delivered substantial performance improvements given large numbers of rays.

The Embree system, described in a paper by Wald et al. (2014), is a high-performance open source rendering system that supports both packet tracing and highly efficient traversal of single rays. See also the paper by Benthin et al. (2011) on the topic of finding a balance between these two approaches.

`pbrt` is very much a “one ray at a time” ray tracer; if a rendering system can provide many rays for intersection tests at once, a variety of more efficient implementations are possible even beyond packet tracing. For example, Keller and Wächter (2011) and Mora (2011) described algorithms for intersecting a large number of rays against the scene

geometry where there is no acceleration structure at all. Instead, primitives and rays are both recursively partitioned until small collections of rays and small collections of primitives remain, at which point intersection tests are performed. Improvements to this approach were described by Áfra (2012) and Nabata et al. (2013).

17.2.3 RAY-TRACING HARDWARE

Given the widespread success of specialized hardware for triangle rasterization and shading in modern PCs, there has long been interest in designing specialized hardware for ray tracing. The ray-tracing algorithm presents a variety of stages of computation that must be addressed in a complete system, including camera ray generation, construction of the acceleration hierarchy, traversal of the hierarchy, ray-primitive intersections, shading, lighting, and integration calculations.

Early published work in this area includes a paper by Woop et al. (2005), who described the design of a “ray processing unit” (RPU). More recently, Aila and Karras (2010) described general architectural issues related to handling incoherent rays, as are common with global illumination algorithms. Nah et al. (2011) and Lee and collaborators (2013, 2015) have written a series of papers on ray tracing on a mobile GPU architecture, addressing issues including hierarchy traversal, ray generation, intersection calculations, and ray reordering for better memory coherence. See also the paper by Doyle et al. (2013) on SAH BVH construction in specialized hardware.

While there has been substantial research work in this area, unfortunately none of these architectures has made it out to the market in large numbers, though the Caustic ray-tracing architecture (McCombe 2013) has been acquired by a mobile GPU vendor, Imagination Technologies. Plans for products based on an integration of this architecture into a traditional GPU have been announced; we are hopeful that the time for efficient ray-tracing hardware may have arrived.

17.2.4 THE FUTURE

Innovation in high-performance architectures for graphics seems likely to continue in coming years. As CPUs are gradually increasing their SIMD width and adding more processing cores, becoming more similar to throughput processors, throughput processors are adding support for task parallelism and improving their performance on more irregular workloads than purely data-parallel ones. Whether the computer system of the future is a heterogeneous collection of both types of processing cores or whether there is a middle ground with a single type of processor architecture that works well for a range of applications remains an open question.

The role of specialized fixed-function graphics hardware in future systems is likely to become increasingly important; fixed-function hardware is generally substantially more power-efficient than programmable hardware. As the critical computational kernels of future graphics systems become clear, fixed-function implementations of them may become widespread.

17.3 CONCLUSION

The idea for pbrt was born in October 1999. Over the next five years, it evolved from a system designed only to support the students taking Stanford’s CS348b course to a robust, feature-rich, extensible rendering system. Since its inception, we have learned a great deal about what it takes to build a rendering system that doesn’t just make pretty pictures but is one that other people enjoy using and modifying as well. What has been most difficult, however, was designing a large piece of software that others might enjoy *reading*. This has been a far more challenging (and rewarding) task than implementing any of the rendering algorithms themselves.

After its first publication, the book enjoyed widespread adoption in advanced graphics courses worldwide, which we found very gratifying. We were unprepared, however, for the impact that pbrt has had on rendering research. Writing a ray tracer from scratch is a formidable task (as so many students in undergraduate graphics courses can attest to), and creating a robust physically based renderer is much harder still. We are proud that pbrt has lowered the barrier to entry for aspiring researchers in rendering, making it easier for researchers to experiment with and demonstrate the value of new ideas in rendering. We continue to be delighted to see papers in SIGGRAPH, the Eurographics Rendering Symposium, High Performance Graphics, and other graphics research venues that either build on pbrt to achieve their goals, or compare their images to pbrt as “ground truth.”

More recently, we have been delighted again to see the rapid adoption of physically based approaches in practice for offline rendering and, recently as of this writing, games and interactive applications. Though we are admittedly unusual folk, it’s a particular delight to see incredible graphics on a screen and marvel at the billions of pseudo-random (or quasi-random) samples, billions of rays traced, and the complex mathematics that went into each image passing by.

We would like to sincerely thank everyone who has built upon this work for their own research, to build a new curriculum, to create amazing movies or games, or just to learn more about rendering. We hope that this new edition continues to serve the graphics community in the same way that its predecessors were able to.

UTILITIES

In addition to all of the graphics-related code presented thus far, pbrt makes use of a number of general utility routines and classes. Although these are key to pbrt’s operation, it is not necessary to understand their implementation in detail in order to work with the rest of the system. This appendix describes the interfaces to these routines, including those that handle error reporting, memory management, support for parallel execution on multiple CPU cores, and other basic infrastructure. The implementations of some of this functionality—the parts that are interesting enough to be worth delving into—are also discussed.

A.1 MAIN INCLUDE FILE

The `core/pbrt.h` file is included by all other source files in the system. It contains all global function declarations and inline functions, a few macros and numeric constants, and other globally accessible data. All files that include `pbrt.h` get a number of other included header files from `pbrt.h`. This simplifies creation of new source files, almost all of which will want access to these extra headers. However, in the interest of compile time efficiency, we keep the number of these automatically included files to a minimum; the ones here are necessary for almost all modules.

```
{Global Include Files} ≡
    #include <algorithm>
    #include <cinttypes>
    #include <cmath>
    #include <iostream>
    #include <limits>
    #include <memory>
    #include <string>
    #include <vector>
```

Almost all floating-point values in pbrt are declared as `Floats`. (The only exception is a few cases where a 32-bit `float` or a 64-bit `double` is specifically needed (e.g., when saving binary values to files). Whether a `Float` is actually a `float` or a `double` is determined at compile time with the `PBRT_FLOAT_AS_DOUBLE` macro; this makes it possible to build versions of pbrt using either representation. 32-bit `floats` almost always have sufficient precision for ray tracing, but it's helpful to be able to switch to `double` for numerically tricky situations as well as to verify that rounding error with `floats` isn't causing errors for a given scene.

```
(Global Forward Declarations) +≡
#ifndef PBRT_FLOAT_AS_DOUBLE
typedef double Float;
#else
typedef float Float;
#endif // PBRT_FLOAT_AS_DOUBLE
```

A.1.1 UTILITY FUNCTIONS

A few short mathematical functions are useful throughout pbrt.

Clamping

`Clamp()` clamps the given value `val` to lie between the values `low` and `high`. For convenience `Clamp()` allows the types of the values giving the extent to be different than the type being clamped (but its implementation requires that implicit conversion is legal to the type being clamped). By being implemented this way, the implementation allows calls like `Clamp(floatValue, 0, 1)` which would otherwise be disallowed by C++'s template type resolution rules.

```
(Global Inline Functions) +≡
template <typename T, typename U, typename V>
inline T Clamp(T val, U low, V high) {
    if (val < low) return low;
    else if (val > high) return high;
    else return val;
}
```

Modulus

`Mod()` computes the remainder of a/b . pbrt has its own version of this (rather than using `%`) in order to provide the behavior that the modulus of a negative number is always positive. Starting with C++11, the behavior of `%` has been specified to return a negative value in this case, so that the identity $(a/b)*b + a\%b == a$ holds.

```
(Global Inline Functions) +≡
template <typename T> inline T Mod(T a, T b) {
    T result = a - (a/b) * b;
    return (T)((result < 0) ? result + b : result);
}
```

`Clamp()` [1062](#)

`Float` [1062](#)

`Mod()` [1062](#)

A specialization for `Floats` calls out to the corresponding standard library function.

```
<Global Inline Functions> +≡
    template <> inline Float Mod(Float a, Float b) {
        return std::fmod(a, b);
    }
```

Useful Constants

A number of constants, most of them related to π , are used enough that it's worth having them easily available.

```
<Global Constants> +≡
    static const Float Pi      = 3.14159265358979323846;
    static const Float InvPi   = 0.31830988618379067154;
    static const Float Inv2Pi  = 0.15915494309189533577;
    static const Float Inv4Pi  = 0.07957747154594766788;
    static const Float PiOver2 = 1.57079632679489661923;
    static const Float PiOver4 = 0.78539816339744830961;
    static const Float Sqrt2   = 1.41421356237309504880;
```

Converting between Angle Measures

Two simple functions convert from angles expressed in degrees to radians, and vice versa:

```
<Global Inline Functions> +≡
    inline Float Radians(Float deg) {
        return (Pi / 180) * deg;
    }
    inline Float Degrees(Float rad) {
        return (180 / Pi) * rad;
    }
```

Base-2 Operations

Because the math library doesn't provide a base-2 logarithm function, we provide one here, using the identity $\log_2(x) = \log x / \log 2$.

```
<Global Inline Functions> +≡
    inline Float Log2(Float x) {
        const Float invLog2 = 1.442695040888963387004650940071;
        return std::log(x) * invLog2;
    }
```

It's also useful to be able to compute an integer base-2 logarithm. Rather than computing an (expensive) floating-point logarithm and converting to an integer, it's much more efficient to count the number of leading zeros up to the first one in the 32-bit binary representation of the value and then subtract this value from 31, which gives the index of the first bit set, which is in turn the integer base-2 logarithm. (This efficiency comes in part from the fact that most CPUs have an instruction to count these zeros.)

Float 1062

Pi 1063

The code here uses the `_builtin_clz()` intrinsic, which is available in the `g++` and `clang` compilers; `_BitScanReverse()` is used to implement similar functionality with MSVC in code that isn't shown here.

```
(Global Inline Functions) +≡
inline int Log2Int(uint32_t v) {
    return 31 - __builtin_clz(v);
}
```

There are clever tricks that can be used to efficiently determine if a given integer is an exact power of 2, or round an integer up to the next higher (or equal) power of 2. (It's worthwhile to take a minute and work through for yourself how these two functions work.)

```
(Global Inline Functions) +≡
template <typename T> inline bool IsPowerOf2(T v) {
    return v && !(v & (v - 1));
}
```

```
(Global Inline Functions) +≡
inline int32_t RoundUpPow2(int32_t v) {
    v--;
    v |= v >> 1;      v |= v >> 2;
    v |= v >> 4;      v |= v >> 8;
    v |= v >> 16;
    return v+1;
}
```

A variant of RoundUpPow2() for int64_t is also provided but isn't included in the text here.

Some of the low-discrepancy sampling code in Chapter 7 needs to efficiently count the number of trailing zeros in the binary representation of a value; CountTrailingZeros() is a wrapper around a compiler-specific intrinsic that maps to a single instruction on most architectures.

```
(Global Inline Functions) +≡
inline int CountTrailingZeros(uint32_t v) {
    return __builtin_ctz(v);
}
```

Interval Search

FindInterval() is a helper function that emulates the behavior of std::upper_bound(), but uses a function object to get values at various indices instead of requiring access to an actual array. This way, it becomes possible to bisect arrays that are procedurally generated, such as those interpolated from point samples. The implementation here also adds some bounds checking for corner cases (e.g., making sure that a valid interval is selected even in the case the predicate evaluates to true or false for all entries), which would normally have to follow a call to std::upper_bound().

```

⟨Global Inline Functions⟩ +≡
    template <typename Predicate> int FindInterval(int size,
        const Predicate &pred) {
        int first = 0, len = size;
        while (len > 0) {
            int half = len >> 1, middle = first + half;
            ⟨Bisect range based on value of pred at middle 1065⟩
        }
        return Clamp(first - 1, 0, size - 2);
    }

⟨Bisect range based on value of pred at middle⟩ ≡
    if (pred(middle)) {
        first = middle + 1;
        len -= half + 1;
    } else
        len = half;

```

1065

A.1.2 PSEUDO-RANDOM NUMBERS

pbrt uses an implementation of the PCG pseudo-random number generator (O’Neill 2014) to generate pseudo-random numbers. This generator is one of the best random number generators currently known. Not only does it pass a variety of rigorous statistical tests that have been the bane of earlier pseudo-random number generators, but its implementation is also extremely efficient.

We wrap its implementation in a small random number generator class, `RNG`. Doing so allows us to use it with slightly less verbose calls throughout the rest of the system. Random number generator implementation is an esoteric art; therefore, we will not include or discuss the implementation here but will describe the APIs provided.

The `RNG` class provides two constructors. The first, which takes no arguments, sets the internal state to reasonable defaults. The second takes a single argument that selects a sequence of pseudo-random values.

The PCG random number generator actually allows the user to provide two 64-bit values to configure its operation: one chooses from one of 2^{63} different sequences of 2^{64} random numbers, while the second effectively selects a starting point within such a sequence. Many pseudo-random number generators only allow this second form of configuration, which alone isn’t as good: having independent non-overlapping sequences of values rather than different starting points in a single sequence provides greater non-uniformity in the generated values.

For pbrt’s needs, selecting different sequences is sufficient, so the `RNG` implementation doesn’t provide a mechanism to also select the starting point within a sequence.

```

RNG 1065
RNG::SetSequence() 1066
⟨RNG Public Methods⟩ ≡
    RNG();
    RNG(uint64_t sequenceIndex) { SetSequence(sequenceIndex); }

```

RNGs shouldn't be used in `pbrt` without either providing an initial sequence index via the constructor or a call to the `SetSequence()` method; otherwise there's risk that different parts of the system will inadvertently use correlated sequences of pseudo-random values, which in turn could cause surprising errors.

(RNG Public Methods) +≡

```
void SetSequence(uint64_t sequenceIndex);
```

There are two variants of the `UniformUInt32()` method. The first returns a pseudo-random number in the range $[0, 2^{32} - 1]$.

(RNG Public Methods) +≡

```
uint32_t UniformUInt32();
```

The second returns a value uniformly distributed in the range $[0, b - 1]$ given a bound b . The last two versions of `pbrt` effectively used `UniformUInt32() % b` for this second computation. That approach is subtly flawed—in the case that b doesn't evenly divide 2^{32} , then there is slightly higher probability of choosing values in the sub-range $[0, 2^{32} \bmod b]$.

The implementation here first computes the above remainder $2^{32} \bmod b$ efficiently using only 32 bit arithmetic and stores it in the variable `threshold`. Then, if the pseudo-random value returned by `UniformUInt32()` is less than `threshold`, it is discarded and a new value is generated. The resulting distribution of values has a uniform distribution after the modulus operation, giving a uniformly distributed sample value.

(RNG Public Methods) +≡

```
uint32_t UniformUInt32(uint32_t b) {
    uint32_t threshold = (~b + 1u) % b;
    while (true) {
        uint32_t r = UniformUInt32();
        if (r >= threshold)
            return r % b;
    }
}
```

`UniformFloat()` generates a pseudo-random floating-point number in the half-open interval $[0, 1)$.

(RNG Public Methods) +≡

```
Float UniformFloat() {
    return std::min(OneMinusEpsilon, UniformUInt32() * 0x1p-32f);
}
```

A.2 IMAGE FILE INPUT AND OUTPUT

Many image file formats have been developed over the years, but for `pbrt`'s purposes we are mainly interested in those that support imagery represented by floating-point pixel values. In particular, the images generated by `pbrt` will often have a large dynamic range; such formats are crucial for being able to store the computed radiance values directly.

`Float` 1062

`OneMinusEpsilon` 417

`RNG::UniformUInt32()` 1066

Legacy image file formats, such as those that store 8 bits of data for red, green, and blue components to represent colors in the range [0, 1], aren't a good fit for physically based rendering needs.

pbrt supports two floating-point image file formats: OpenEXR and PFM. OpenEXR is a floating-point file format originally designed at Industrial Light and Magic for use in movie productions (Kainz et al. 2004). We chose this format because it has a clean design, is easy to use, and has first-class support for floating-point image data. Libraries that read and write OpenEXR images are freely available, and support for the format is available in many other tools.

PFM is a floating-point format based on an extension to the PPM file format; it is very easily read and written, though it isn't as widely supported as OpenEXR. Unlike OpenEXR, it doesn't support compression, so files may be fairly large.

For convenience, pbrt also has support to read and write TGA format files as well as support to read and write PNG images. Neither of these is a high-dynamic-range format like OpenEXR, but both are convenient, especially as input formats for low-dynamic-range texture maps.

The `ReadImage()` function takes the filename to read from and a pointer to a `Point2i` that will be initialized with the image resolution. It returns a pointer to the start of a freshly allocated array of `RGBSpectrum` objects. It will read the given file as an OpenEXR, PFM, PNG, or TGA file, depending on the suffix of the filename.

```
(ImageIO Declarations) ≡  
std::unique_ptr<RGBSpectrum[]> ReadImage(const std::string &name,  
    Point2i *resolution);
```

`ReadImage()` uses `RGBSpectrum` for the return values—not `Spectrum`. The primary client of this function is the image texture mapping code in pbrt, which stores texture maps as `RGBSpectrum` values, even when pbrt is compiled to do full-spectral rendering, so returning `RGBSpectrum` values is a natural approach. (We also made this decision under the expectation that the image files being read would be in RGB or another three-channel format, so that returning RGB values wouldn't discard spectral information; if calling code wants to store full `Spectrum` values, then it can convert from RGB to the full-spectral representation itself.) If pbrt was extended to support a full-spectral input image format for textures, then a variant of this function that did return `Spectrum` values would be advisable.

The `WriteImage()` function takes a filename to be written, a pointer to the beginning of the pixel data, and information about the resolution of the image. The pixel data should be organized as interleaved `RGBRGB . . .` values. Like `ReadImage()`, it uses the suffix of the given filename to determine which image format to use.

With `WriteImage()`, it's possible to specify that the pixels being written represent a sub-region of a larger image. Some image formats (e.g., OpenEXR) can record this information in the image file header, which in turn makes it easy to assemble separately rendered subimages into a single image. The `totalResolution` parameter gives the total resolution

`Point2i` 68

`ReadImage()` 1067

`RGBSpectrum` 332

`Spectrum` 315

`WriteImage()` 1068

of the overall image that the given pixel values are part of, and `outputBounds` gives the pixel bounding box that the given pixels cover. `outputBounds` should be within the range $(0, 0) \rightarrow \text{totalResolution}$ and the number of RGB pixel values pointed to by `rgb` should be equal to `outputBounds.Area()`.

If a non-floating-point image format is being used for output, pixel values are converted to the sRGB representation (Section 10.4.1) and clamped to the range [0, 255] before being written to the file.

```
(ImageIO Declarations) +≡
void WriteImage(const std::string &name, const Float *rgb,
    const Bounds2i &outputBounds, const Point2i &totalResolution);
```

We will not show the code that interfaces with the various image-writing libraries or the code that implements file-format-specific I/O. This code can be found in the file `core/imageio.cpp` and the directory `ext/`.¹

A.3 COMMUNICATING WITH THE USER

A number of functions and classes are useful to mediate communicating information to the user. In addition to consolidating functionality like printing progress bars, hiding user communication behind a small API like the one here also permits easy modification of the communication mechanisms. For example, if `pbrt` were embedded in an application that had a graphical user interface, errors might be reported via a dialog box or a routine provided by the parent application. If `printf()` calls were strewn throughout the system, it would be more difficult to make the two systems work together well.

A.3.1 ERROR REPORTING

`pbrt` provides four functions for reporting anomalous conditions. In order of increasing severity, they are `Info()`, `Warning()`, `Error()`, and `Severe()`. These functions are defined in the files `core/error.h` and `core/error.cpp`. All of them take a formatting string as their first argument and a variable number of additional arguments providing values for the format. The syntax is identical to that used by the `printf` family of functions. For example, if the variable `rayNum` has type `int`, then the following call could be made:

```
Info("Now tracing ray number %d", rayNum);
```

`core/pbrt.h` includes this header file, as these functions are useful to have available in almost all parts of the system.

```
(Global Include Files) +≡
#include "error.h"
```

¹ The TGA implementation is based on open source TGA code by Emil Miklic; Jiawen “Kevin” Chen provided the PFM reader and writer; PNG files are handled using the `lodepng` library.

Bounds2i 76

Float 1062

Point2i 68

We will not show the implementation of these functions here because they are a straightforward application of the C++ variable argument processing functions that in turn calls a common function to print the full error string. For sufficiently severe errors, the program aborts.

`pbrt` also has its own version of the standard `assert()` macro, named `Assert()`. It checks that the given expression's value evaluates to true; if not, `Severe()` is called with information about the location of the assertion failure. `Assert()` is used for basic sanity checks where failure indicates little possibility of recovery. In general, assertions should be used to detect internal bugs in the code, not expected error conditions (such as invalid scene file input), because the message printed will likely be cryptic to anyone other than the developer.

```
(Global Inline Functions) +≡
#ifndef NDEBUG
#define Assert(expr) ((void)0)
#else
#define Assert(expr) \
  ((expr) ? (void)0 : \
   Severe("Assertion \"%s\" failed in %s, line %d", \
          #expr, __FILE__, __LINE__))
#endif // NDEBUG
```

A.3.2 REPORTING PROGRESS

The `ProgressReporter` class gives the user feedback about how much of a task has been completed and how much longer it is expected to take. For example, implementations of the various `Integrator::Render()` methods generally use a `ProgressReporter` to show rendering progress. The implementation prints a row of plus signs, the elapsed time, and the estimated remaining time. Its implementation is in the files `core/progressreporter.h` and `core/progressreporter.cpp`.

The constructor takes the total number of units of work to be done (e.g., the total number of camera rays that will be traced) and a short string describing the task being performed.

```
(ProgressReporter Public Methods) ≡
ProgressReporter(int64_t totalWork, const std::string &title);
```

Once the `ProgressReporter` has been created, each call to its `Update()` method signifies that one unit of work has been completed. An optional integer value can be passed to indicate that multiple units have been done.

```
Assert() 1069
Integrator::Render() 25
ProgressReporter 1069
ProgressReporter::Done()
  1070
Severe() 1068
```

```
(ProgressReporter Public Methods) +≡
void Update(int64_t num = 1);
```

The `ProgressReporter::Done()` method should be called when all of the work being measured is complete; in turn, it lets the user know that the task is complete.

(ProgressReporter Public Methods) +≡
`void Done();`

A.3.3 SIMPLE FLOAT FILE READER

A number of places in the `pbrt` code need to read text files that store a series of floating-point values. Examples include the code that reads measured spectral distribution data and the code that reads lens description files in Section 6.4. Both use the `ReadFloatFile()` function, which parses text files of whitespace-separated numbers, returning the values found in the given vector. The parsing code ignores all text after a hash mark (#) to the end of its line to allow comments.

(floatfile.h) ≡*
`bool ReadFloatFile(const char *filename, std::vector<Float> *values);`

A.4 MEMORY MANAGEMENT

Memory management is often a complex issue in a system written in a language without garbage collection. The situation is mostly simple in `pbrt`, since most dynamic memory allocation is done as the scene description file is parsed, and most of this memory remains in use until rendering is finished. Nevertheless, there are a few issues related to memory management—most of them performance related—that warrant classes and utility routines to address them.

A.4.1 VARIABLE STACK ALLOCATION

Sometimes it is necessary to allocate a variable amount of memory that will be used temporarily in a single function but isn’t needed after the function returns. If only a small amount of memory is needed, the overhead of `new` and `delete` (or `malloc()` and `free()`) may be high relative to the amount of actual computation being done. Instead, it is frequently more efficient to use `alloca()`, which allocates memory on the stack with just a few machine instructions. This memory is automatically deallocated when the function exits, which also saves bookkeeping work in the routine that uses it.

`alloca()` is an extremely useful tool, but there are two pitfalls to be aware of when using it. First, because the memory is deallocated when the function that called `alloca()` returns, the pointer must not be returned from the function or stored in a data structure with a longer lifetime than the function that allocated it. (However, the pointer may be passed to functions called by the allocating function.) Second, stack size is limited, and so `alloca()` shouldn’t be used for more than a few kilobytes of storage. Unfortunately, there is no way to detect the error condition when more space is requested from `alloca()` than is available on the stack, so it’s important to be conservative with its use.

`pbrt` provides a macro that makes it easy to allocate space for a given number of objects of a given type.²

Float 1062

² A moment’s thought should make clear why it’s not possible to implement this functionality with an `inline` function.

{Global Macros} ≡

```
#define ALLOCA(TYPE, COUNT) (TYPE *)alloca((COUNT) * sizeof(TYPE))
```

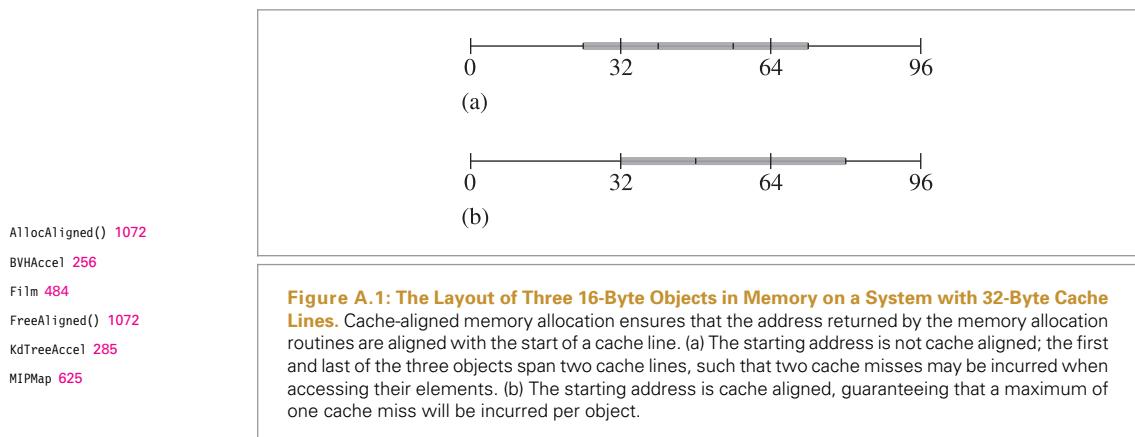
A.4.2 CACHE-FRIENDLY MEMORY USAGE

The speed at which memory can respond to read requests has historically been getting faster at a rate of roughly 10% per year, while the computational capabilities of modern CPUs have been growing much more quickly. As such, a CPU will typically have to wait a hundred or so execution cycles to read from main memory. The CPU is usually idle for much of this time, so a substantial amount of its computational potential may be lost.

One of the most effective techniques to address this problem is the judicious use of small, fast cache memory located in the CPU itself. The cache holds recently accessed data and is able to service memory requests much faster than main memory, thus greatly reducing the frequency of stalls in the CPU.

Because of the high penalty for accessing main memory, designing algorithms and data structures that make good use of the cache can substantially improve overall system performance. This section will discuss general programming techniques for improving cache performance. These techniques are used in many parts of pbrt, particularly the KdTreeAccel, BVHAccel, MIPMap, and Film. We assume that the reader has a basic familiarity with computer architecture and caching technology; readers needing a review are directed to a computer architecture text such as Hennessy and Patterson (1997). In particular, the reader should be generally familiar with topics like cache lines, cache associativity, and the difference between compulsory, capacity, and conflict misses.

One easy way to reduce the number of cache misses incurred by pbrt is to make sure that some key memory allocations are aligned with the blocks of memory that the cache manages. (pbrt’s overall performance was improved by approximately 3% when allocation for the kd-tree accelerator in Section 4.4 was rewritten to use cache-aligned allocation.) Figure A.1 illustrates the basic technique. The AllocAligned() and FreeAligned() functions



provide an interface to allocate and release cache-aligned memory blocks. If the preprocessor constant `PBRT_L1_CACHE_LINE_SIZE` is not set, a default cache line size of 64 bytes is used, which is representative of many current architectures.

```
(Global Constants) +≡
#ifndef PBRT_L1_CACHE_LINE_SIZE
#define PBRT_L1_CACHE_LINE_SIZE 64
#endif
```

Unfortunately there aren't portable methods to allocate memory aligned to a particular granularity. Therefore `AllocAligned()` must call various operating-system-specific functions to do these allocations.

```
(Memory Allocation Functions) ≡
void *AllocAligned(size_t size) {
    #if defined(PBRT_IS_WINDOWS)
        return _aligned_malloc(size, PBRT_L1_CACHE_LINE_SIZE);
    #elif defined(PBRT_IS_OPENBSD) || defined(PBRT_IS OSX)
        void *ptr;
        if (posix_memalign(&ptr, PBRT_L1_CACHE_LINE_SIZE, size) != 0)
            ptr = nullptr;
        return ptr;
    #else
        return memalign(PBRT_L1_CACHE_LINE_SIZE, size);
    #endif
}
```

A convenience routine is also provided for allocating a collection of objects so that code like `AllocAligned<Foo>(n)` can be written to allocate an array of `n` instances of type `Foo`.

```
(Memory Declarations) +≡
template <typename T> T *AllocAligned(size_t count) {
    return (T *)AllocAligned(count * sizeof(T));
}
```

The routine for freeing aligned memory calls the corresponding operating-system-specific routine. We won't include its implementation here.

```
(Memory Declarations) +≡
void FreeAligned(void *);
```

Another family of techniques for improving cache performance is based on reorganizing data structures themselves. For example, using bit fields to reduce the size of a frequently used data structure can be helpful. This approach improves the *spatial locality* of memory access at run time, since code that accesses multiple packed values won't incur more than one cache miss to get them all. Furthermore, by reducing the overall size of the structure, this technique can reduce capacity misses if fewer cache lines are consequently needed to store the structure.

If not all of the elements of a structure are frequently accessed, there are a few possible strategies to improve cache performance. For example, if the structure has a size of 128 bytes and the computer has 64-byte cache lines, two cache misses may be needed to access it. If the commonly used fields are collected into the first 64 bytes rather than being spread throughout, then no more than one cache miss will be incurred when only those fields are needed (Truong, Bodin, and Seznec 1998).

A related technique is *splitting*, where data structures are split into “hot” and “cold” parts, each stored in separate regions of memory. For example, given an array of some structure type, we can split it into two arrays, one for the more frequently accessed (or “hot”) portions and one for the less frequently accessed (or “cold”) portions. This way, cold data doesn’t displace useful information in the cache except when it is actually needed.

Cache-friendly programming is a complex engineering task, and we will not cover all the variations here. Readers are directed to the “Further Reading” section of this appendix for more information.

A.4.3 ARENA-BASED ALLOCATION

Conventional wisdom says that the system’s memory allocation routines (e.g., `malloc()` and `new()`) are slow and that custom allocation routines for objects that are frequently allocated or freed can provide a measurable performance gain. However, this conventional wisdom seems to be wrong. Wilson et al. (1995), Johnstone and Wilson (1999), and Berger, Zorn, and McKinley (2001, 2002) all investigated the performance impact of memory allocation in real-world applications and found that custom allocators almost always result in *worse* performance than a well-tuned generic system memory allocation, in both execution time and memory use.

One type of custom allocation technique that has proved to be useful in some cases is *arena-based allocation*, which allows the user to quickly allocate objects from a large contiguous region of memory. In this scheme, individual objects are never explicitly freed; the entire region of memory is released when the lifetime of all of the allocated objects ends. This type of memory allocator is a natural fit for many of the objects in `pbrt`.

There are two main advantages to arena-based allocation. First, allocation is extremely fast, usually just requiring a pointer increment. Second, it can improve locality of reference and lead to fewer cache misses, since the allocated objects are contiguous in memory. A more general dynamic memory allocator will typically prepend a bookkeeping structure to each block it returns, which adversely affects locality of reference.

`pbrt` provides the `MemoryArena` class to implement this approach; it supports variable-sized allocation from the arena.

The `MemoryArena` quickly allocates memory for objects of variable size by handing out pointers into a preallocated block. It does not support freeing of individual blocks of memory, only freeing of all of the memory in the arena at once. Thus, it is useful when a number of allocations need to be done quickly and all of the allocated objects have similar lifetimes.

```
(Memory Declarations) +≡
class MemoryArena {
public:
    (MemoryArena Public Methods 1074)
private:
    (MemoryArena Private Data 1074)
};
```

MemoryArena allocates memory in chunks of size `MemoryArena::blockSize`, the value of which is set by a parameter passed to the constructor. If no value is provided to the constructor, a default of 256 kB is used.

(MemoryArena Public Methods) ≡ 1074

```
MemoryArena(size_t blockSize = 262144) : blockSize(blockSize) {}
```

(MemoryArena Private Data) ≡ 1074

```
const size_t blockSize;
```

The implementation maintains a pointer to the current block of memory, `currentBlock`, and the offset of the first free location in the block, `currentPos`. `currentAllocSize` stores the total size of the `currentBlock` allocation; it generally has the value `blockSize` but is larger in certain cases (discussed in the following).

(MemoryArena Private Data) +≡ 1074

```
size_t currentBlockPos = 0, currentAllocSize = 0;
uint8_t *currentBlock = nullptr;
```

To service an allocation request, the allocation routine first rounds the requested amount of memory up so that it meets the computer's word alignment requirements.³ The routine then checks to see if the current block has enough space to handle the request, allocating a new block if necessary. Finally, it returns the pointer and updates the current block offset.

(MemoryArena Public Methods) +≡ 1074

```
void *Alloc(size_t nBytes) {
    (Round up nBytes to minimum machine alignment 1075)
    if (currentBlockPos + nBytes > currentAllocSize) {
        (Add current block to usedBlocks list 1075)
        (Get new block of memory for MemoryArena 1075)
    }
    void *ret = currentBlock + currentBlockPos;
    currentBlockPos += nBytes;
    return ret;
}
```

Most modern computer architectures impose alignment requirements on the positioning of objects in memory. For example, it is frequently a requirement that `float` values be

MemoryArena 1074
`MemoryArena::blockSize` 1074
`MemoryArena::currentAllocSize`
1074
`MemoryArena::currentBlock` 1074
`MemoryArena::currentBlockPos` 1074

³ Some systems (such as those based on Intel® processors) can handle non-word-aligned memory accesses, but this is usually substantially times slower than word-aligned memory reads or writes. Other architectures do not support this at all and will generate a bus error if a nonaligned access is performed.

stored at memory locations that are word aligned. To be safe, the implementation always hands out 16-byte-aligned pointers (i.e., their address is a multiple of 16).

(Round up nBytes to minimum machine alignment) ≡

```
nBytes = ((nBytes + 15) & (~15));
```

1074

If a new block of memory must be dynamically allocated to service an allocation request, the MemoryArena stores the pointer to the current block of memory in the usedBlocks list so that it is not lost. Later, when MemoryArena::Reset() is called, it will be able to reuse the block for the next series of allocations.

(Add current block to usedBlocks list) ≡

```
if (currentBlock) {
    usedBlocks.push_back(std::make_pair(currentAllocSize, currentBlock));
    currentBlock = nullptr;
}
```

1074

MemoryArena uses two linked lists to hold pointers to blocks of memory that have been fully used as well as available blocks that were previously allocated but aren't currently in use.

(MemoryArena Private Data) +≡

```
std::list<std::pair<size_t, uint8_t *>> usedBlocks, availableBlocks;
```

1074

If a block of memory of suitable size isn't available from availableBlocks, a new one is allocated.

(Get new block of memory for MemoryArena) ≡

(Try to get memory block from availableBlocks 1075)

```
if (!currentBlock) {
    currentAllocSize = std::max(nBytes, blockSize);
    currentBlock = AllocAligned<uint8_t>(currentAllocSize);
}
currentBlockPos = 0;
```

1074

The allocation routine first checks to see if there are any already allocated free blocks in availableBlocks.

(Try to get memory block from availableBlocks) ≡

```
for (auto iter = availableBlocks.begin(); iter != availableBlocks.end();
     ++iter) {
    if (iter->first >= nBytes) {
        currentAllocSize = iter->first;
        currentBlock = iter->second;
        availableBlocks.erase(iter);
        break;
    }
}
```

1075

AllocAligned() 1072
 MemoryArena 1074
 MemoryArena::blockSize 1074
 MemoryArena::currentAllocSize 1074
 MemoryArena::currentBlock 1074
 MemoryArena::Reset() 1076
 MemoryArena::usedBlocks 1075

The MemoryArena also provides a convenience template method to allocate an array of objects of the given type.

```
(MemoryArena Public Methods) +≡ 1074
template<typename T> T *Alloc(size_t n = 1, bool runConstructor = true) {
    T *ret = (T *)Alloc(n * sizeof(T));
    if (runConstructor)
        for (size_t i = 0; i < n; ++i)
            new (&ret[i]) T();
    return ret;
}
```

When the user is done with all of the memory, the arena resets its offset in the current block and moves all of the memory from the `usedBlocks` list onto the `availableBlocks` list.

```
(MemoryArena Public Methods) +≡ 1074
void Reset() {
    currentBlockPos = 0;
    availableBlocks.splice(availableBlocks.begin(), usedBlocks);
}
```

A.4.4 BLOCKED 2D ARRAYS

In C++, 2D arrays are arranged in memory so that entire rows of values are contiguous in memory, as shown in Figure A.2(a). This is not always an optimal layout, however; for such an array indexed by (u, v) , nearby (u, v) array positions will often map to distant memory locations. For all but the smallest arrays, the adjacent values in the v direction will be on different cache lines; thus, if the cost of a cache miss is incurred to reference a value at a particular location (u, v) , there is no chance that handling that miss will also load into memory the data for values $(u, v + 1)$, $(u, v - 1)$, and so on. Thus, spatially coherent array indices in (u, v) do not necessarily lead to the spatially coherent memory access patterns that modern memory caches depend on.

To address this problem, the `BlockedArray` template implements a generic 2D array of values, with the items ordered in memory using a *blocked* memory layout, as shown in Figure A.2(b). The array is subdivided into square blocks of a small fixed size that is a power of 2. Each block is laid out row by row, as if it were a separate 2D C++ array. This organization substantially improves the memory coherence of 2D array references in practice and requires only a small amount of additional computation to determine the memory address for a particular position (Lam, Rothberg, and Wolf 1991).

To ensure that the block size is a power of 2, the caller specifies its logarithm (base 2), which is given by the template parameter `logBlockSize`.

```
(Memory Declarations) +≡
template <typename T, int logBlockSize> class BlockedArray {
public:
    BlockedArray Public Methods 1077
private:
    BlockedArray Private Data 1078
};
```

BlockedArray 1076
MemoryArena::currentBlockPos 1074

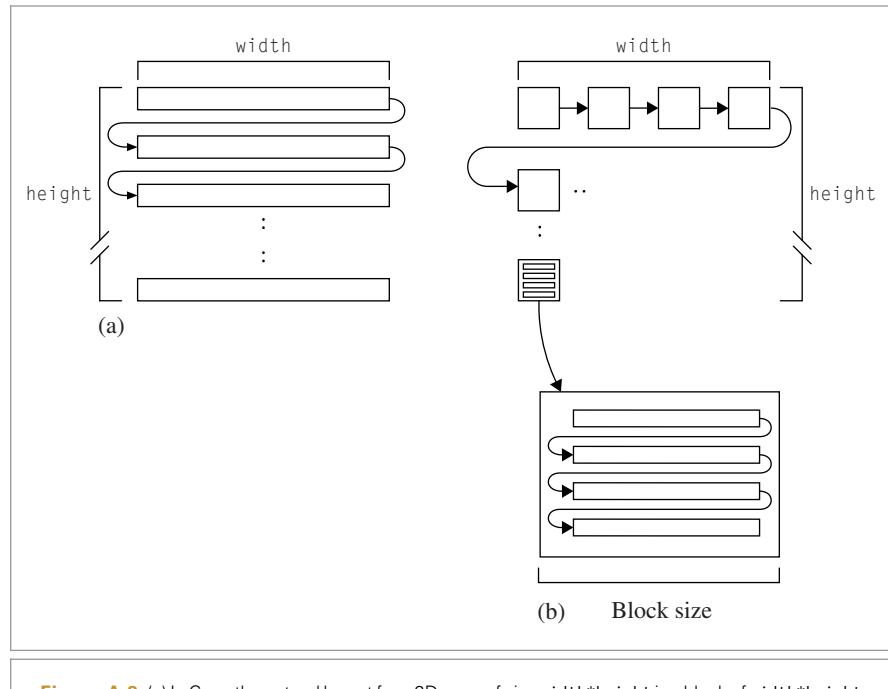


Figure A.2: (a) In C++, the natural layout for a 2D array of size $\text{width} \times \text{height}$ is a block of $\text{width} \times \text{height}$ entries, where the (u, v) array element is at the $u+v \times \text{width}$ offset. (b) A blocked array has been split into smaller square blocks, each of which is laid out linearly. Although it is slightly more complex to find the memory location associated with a given (u, v) array position in the blocked scheme, the improvement in cache performance due to more coherent memory access patterns often more than makes up for this in overall faster performance.

The constructor allocates space for the array and optionally initializes its values from a pointer to a standard C++ array. Because the array size may not be an exact multiple of the block size, it may be necessary to round up the size in one or both directions to find the total amount of memory needed for the blocked array. The `BlockedArray::RoundUp()` method rounds both dimensions up to be a multiple of the block size.

(BlockedArray Public Methods) ≡

1076

```

    BlockedArray(int uRes, int vRes, const T *d = nullptr)
        : uRes(uRes), vRes(vRes), uBlocks(RoundUp(uRes) >> logBlockSize) {
        int nAlloc = RoundUp(uRes) * RoundUp(vRes);
        data = AllocAligned<T>(nAlloc);
        for (int i = 0; i < nAlloc; ++i)
            new (&data[i]) T();
        if (d)
            for (int v = 0; v < vRes; ++v)
                for (int u = 0; u < uRes; ++u)
                    (*this)(u, v) = d[v * uRes + u];
    }
}

```

AllocAligned() 1072
 BlockedArray 1076
 BlockedArray::RoundUp() 1078

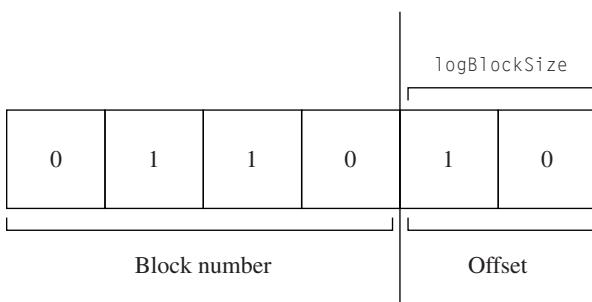


Figure A.3: Given a binary array coordinate, the (u, v) block number that it is in can be found by shifting off the `logBlockSize` low-order bits for both u and v . For example, with a `logBlockSize` of 2 and thus a block size of 4, we can see that this correctly maps 1D array positions from 0 to 3 to block 0, 4 to 7 to block 1, and so on. To find the offset within the particular block, it is just necessary to mask off the high-order bits, leaving the `logBlockSize` low-order bits. Because the block size is a power of two, these computations can all be done with efficient bit operations.

(BlockedArray Private Data) \equiv

1076

```
T *data;
const int uRes, vRes, uBlocks;
```

(BlockedArray Public Methods) $+ \equiv$

1076

```
constexpr int BlockSize() const { return 1 << logBlockSize; }
int RoundUp(int x) const {
    return (x + BlockSize() - 1) & ~BlockSize() - 1;
}
```

For convenience, the `BlockedArray` can also report its size in each dimension:

(BlockedArray Public Methods) $+ \equiv$

1076

```
int uSize() const { return uRes; }
int vSize() const { return vRes; }
```

Looking up a value from a particular (u, v) position in the array requires some indexing work to find the memory location for that value. There are two steps to this process: finding which block the value is in and finding its offset within that block. Because the block sizes are always powers of 2, the `logBlockSize` low-order bits in each of the u and v array positions give the offset within the block, and the high-order bits give the block number (Figure A.3).

(BlockedArray Public Methods) $+ \equiv$

1076

```
int Block(int a) const { return a >> logBlockSize; }
int Offset(int a) const { return (a & (BlockSize() - 1)); }
```

Then, given the block number (b_u, b_v) and the offset within the block (o_u, o_v) , it is necessary to compute what memory location this maps to in the blocked array layout. First consider the task of finding the starting address of the block; since the blocks are laid out row by row, this corresponds to the block number $b_u + b_v * uBlocks$, where `uBlocks` is the

`BlockedArray::BlockSize()`
1078

`BlockedArray::logBlockSize`
1076

`BlockedArray::uRes` 1078

`BlockedArray::vRes` 1078

number of blocks in the u direction. Because each block has `BlockSize() * BlockSize()` values in it, the product of the block number and this value gives us the offset to the start of the block. We then just need to account for the additional offset from the start of the block, which is `ou + ov * BlockSize()`.

(BlockedArray Public Methods) +≡

```
T &operator()(int u, int v) {
    int bu = Block(u), bv = Block(v);
    int ou = Offset(u), ov = Offset(v);
    int offset = BlockSize() * BlockSize() * (uBlocks * bv + bu);
    offset += BlockSize() * ov + ou;
    return data[offset];
}
```

1076

A.5 MATHEMATICAL ROUTINES

This section describes a number of useful mathematical functions and classes that support basic operations in pbrt, such as solving small linear systems, manipulating matrices, and linear interpolation.

The `Lerp()` function linearly interpolates between the two provided values.

(Global Inline Functions) +≡

```
inline Float Lerp(Float t, Float v1, Float v2) {
    return (1 - t) * v1 + t * v2;
}
```

A.5.1 SOLVING QUADRATIC EQUATIONS

The `Quadratic()` function finds solutions of the quadratic equation $at^2 + bt + c = 0$; the Boolean return value indicates whether solutions were found.

(Global Inline Functions) +≡

```
inline bool Quadratic(Float a, Float b, Float c, Float *t0, Float *t1) {
    <Find quadratic discriminant 1079>
    <Compute quadratic t values 1080>
}
```

The implementation always uses double-precision floating-point values regardless of the type of `Float` in order to return a result with minimal floating-point error. If the discriminant ($b^2 - 4ac$) is negative, then there are no real roots and the function returns false.

(Find quadratic discriminant) ≡

```
double discrim = (double)b * (double)b - 4 * (double)a * (double)c;
if (discrim < 0) return false;
double rootDiscrim = std::sqrt(discrim);
```

1079

```
BlockedArray::Block() 1078
BlockedArray::BlockSize()
1078
BlockedArray::data 1078
BlockedArray::operator()
1079
BlockedArray::uBlocks 1078
Float 1062
```

The usual version of the quadratic equation can give poor numerical precision when $b \approx \pm\sqrt{b^2 - 4ac}$ due to cancellation error. It can be rewritten algebraically to a more

stable form:

$$t_0 = \frac{q}{a}$$

$$t_1 = \frac{c}{q},$$

where

$$q = \begin{cases} -.5(b - \sqrt{b^2 - 4ac}) & b < 0, \\ -.5(b + \sqrt{b^2 - 4ac}) & \text{otherwise.} \end{cases}$$

(Compute quadratic t values) ≡

1079

```
double q;
if (b < 0) q = -.5 * (b - rootDiscrim);
else q = -.5 * (b + rootDiscrim);
*t0 = q / a;
*t1 = c / q;
if (*t0 > *t1) std::swap(*t0, *t1);
return true;
```

A.5.2 2×2 LINEAR SYSTEMS

There are a number of places throughout pbrt where we need to solve a 2×2 linear system $Ax = B$ of the form

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \end{pmatrix}$$

for values x_0 and x_1 . The `SolveLinearSystem2x2()` routine finds the closed-form solution to such a system. It returns `true` if it was successful and `false` if the determinant of A is very small, indicating that the system is numerically ill-conditioned and either not solvable or likely to have unacceptable floating-point errors. In this case, no solution is returned.

(Matrix4x4 Method Definitions) ≡

```
bool SolveLinearSystem2x2(const Float A[2][2],
    const Float B[2], Float *x0, Float *x1) {
    Float det = A[0][0] * A[1][1] - A[0][1] * A[1][0];
    if (std::abs(det) < 1e-10f)
        return false;
    *x0 = (A[1][1] * B[0] - A[0][1] * B[1]) / det;
    *x1 = (A[0][0] * B[1] - A[1][0] * B[0]) / det;
    if (std::isnan(*x0) || std::isnan(*x1))
        return false;
    return true;
}
```

Float 1062

Matrix4x4 1081

Transform 83

A.5.3 4×4 MATRICES

The `Matrix4x4` structure provides a low-level representation of 4×4 matrices. It is an integral part of the `Transform` class.

```
<Matrix4x4 Declarations> ≡
    struct Matrix4x4 {
        <Matrix4x4 Public Methods 1081>
        float m[4][4];
    };
```

The default constructor, not shown here, sets the matrix to the identity matrix. The `Matrix4x4` implementation also provides constructors that allow the user to pass an array of floats or 16 individual floats to initialize a `Matrix4x4`:

```
<Matrix4x4 Public Methods> ≡ 1081
    Matrix4x4(float mat[4][4]);
    Matrix4x4(float t00, float t01, float t02, float t03,
              float t10, float t11, float t12, float t13,
              float t20, float t21, float t22, float t23,
              float t30, float t31, float t32, float t33);
```

The implementations of operators that test for equality and inequality are straightforward and not included in the text here.

The `Matrix4x4` class supports a few low-level matrix operations. For example, `Transpose()` returns a new matrix that is the transpose of the original matrix.

```
<Matrix4x4 Method Definitions> +≡
    Matrix4x4 Transpose(const Matrix4x4 &m) {
        return Matrix4x4(m.m[0][0], m.m[1][0], m.m[2][0], m.m[3][0],
                         m.m[0][1], m.m[1][1], m.m[2][1], m.m[3][1],
                         m.m[0][2], m.m[1][2], m.m[2][2], m.m[3][2],
                         m.m[0][3], m.m[1][3], m.m[2][3], m.m[3][3]);
    }
```

The product of two matrices M_1 and M_2 is computed by setting the (i, j) th element of the result to the inner product of the i th row of M_1 with the j th column of M_2 .

```
<Matrix4x4 Public Methods> +≡ 1081
    static Matrix4x4 Mul(const Matrix4x4 &m1, const Matrix4x4 &m2) {
        Matrix4x4 r;
        for (int i = 0; i < 4; ++i)
            for (int j = 0; j < 4; ++j)
                r.m[i][j] = m1.m[i][0] * m2.m[0][j] +
                            m1.m[i][1] * m2.m[1][j] +
                            m1.m[i][2] * m2.m[2][j] +
                            m1.m[i][3] * m2.m[3][j];
        return r;
    }
```

Float 1062
Inverse() 1081
`Matrix4x4` 1081
`Matrix4x4::m` 1081

Finally, `Inverse()` returns the inverse of the matrix. The implementation (not shown here) uses a numerically stable Gauss–Jordan elimination routine to compute the inverse.

```
<Matrix4x4 Public Methods> +≡ 1081
    friend Matrix4x4 Inverse(const Matrix4x4 &);
```

A.6 PARALLELISM

Section 1.4 introduced some basic principles of parallel programming and described their application to pbrt. Here, we'll go into more detail about performance issues related to multi-threading as well as describe the implementation of pbrt's `ParallelFor()` function, which is used throughout the system for parallel for loops, where different iterations of the loop can execute concurrently in different threads.

A.6.1 MEMORY COHERENCE MODELS AND PERFORMANCE

Cache coherence is a feature of all modern multicore CPUs; with it, memory writes by one processor are automatically visible to other processors. This is an incredibly useful feature; being able to assume it in the implementation of a system like pbrt is extremely helpful to the programmer. Understanding the subtleties and performance characteristics of this feature is important, however.

One potential issue is that other processors may not see writes to memory in the same order that the processor that performed the writes issued them. This can happen for two main reasons: the compiler's optimizer may have reordered write operations to improve performance, and the CPU hardware may write values to memory in a different order than the stream of executed machine instructions. In the single-threaded case, both of these are innocuous; by design, the compiler and hardware, respectively, ensure that it's impossible for a single thread of execution running the program to detect when these cases happen. This guarantee is not provided for multi-threaded code, however; doing so would impose a significant performance penalty, so hardware architectures leave handling this problem, when it matters, to software.

Memory barrier instructions can be used to ensure that all write instructions before the barrier are visible in memory before any subsequent instructions execute. In practice, we generally don't need to issue memory barrier instructions explicitly, since the thread synchronization calls used to build multi-threaded algorithms take care of this; they are defined to make sure that writes are visible so that if we are coordinating execution between multiple threads using these calls, then they have a consistent view of memory after synchronization points.

Although cache coherence is helpful to the programmer, it can sometimes impose a substantial performance penalty for data that is frequently modified and accessed by multiple processors. Read-only data has little penalty; copies of it can be stored in the local caches of all of the processors that are accessing it, allowing all of them the same performance benefits from the caches as in the single-threaded case. To understand the downside of taking too much advantage of cache coherence for read–write data, it's useful to understand how cache coherence is typically implemented on processors.

CPUs implement a *cache coherence protocol*, which is responsible for tracking the memory transactions issued by all of the processors in order to provide cache coherence. A classic such protocol is *MESI*, where the acronym represents the four states that each cache line can be in. Each processor stores the current state for each cache line in its local caches:

- *Modified*—The current processor has written to the memory location, but the result is only stored in the cache—it's *dirty* and hasn't been written to main memory. No other processor has the location in its cache.
- *Exclusive*—The current processor is the only one with the data from the corresponding memory location in its cache. The value in the cache matches the value in memory.
- *Shared*—Multiple processors have the corresponding memory location in their caches, but they have only performed read operations.
- *Invalid*—The cache line doesn't hold valid data.

At system startup time, the caches are empty and all cache lines are in the invalid state. The first time a processor reads a memory location, the data for that location is loaded into cache and its cache line marked as being in the “exclusive” state. If another processor performs a memory read of a location that is in the “exclusive” state in another cache, then both caches record the state for the corresponding memory location to instead be “shared.”

When a processor writes to a memory location, the performance of the write depends on the state of the corresponding cache line. If it's in the “exclusive” state and already in the writing processor's cache, then the write is cheap; the data is modified in the cache and the cache line's state is changed to “modified.” (If it was already in the “modified” state, then the write is similarly efficient.) In these cases, the value will eventually be written to main memory, at which point the corresponding cache line returns to the “exclusive” state.

However, if a processor writes to a memory location that's in the “shared” state in its cache or is in the “modified” or “exclusive” state in another processor's cache, then expensive communication between the cores is required. All of this is handled transparently by the hardware, though it still has a performance impact. In this case, the writing processor must issue a *read for ownership* (RFO), which marks the memory location as invalid in the caches of any other processors; RFOs can cause stalls of tens or hundreds of cycles—a substantial penalty for a single memory write.

In general, we'd therefore like to avoid the situation of multiple processors concurrently writing to the same memory location as well as unnecessarily reading memory that another processor is writing to. An important case to be aware of is “false sharing,” where a single cache line holds some read-only data and some data that is frequently modified. In this case, even if only a single processor is writing to the part of the cache line that is modified but many are reading from the read-only part, the overhead of frequent RFO operations will be unnecessarily incurred.

A situation where many processors might be concurrently trying to write to the same or nearby memory locations is when image sample values are accumulated into the final image. To ensure that image updates don't pay the RFO cost, each rendering thread in the `ParallelFor()` loop of the `SamplerIntegrator` creates a private `FilmTile` to use for accumulating sample values for the part of the image that it's working on; it is then free to modify the `FilmTile` pixel values without worrying about contention with other threads for those memory locations. Only when a portion of the image is finished is the

tile merged into the main image, thus allowing the overhead of mutual exclusion and RFO operations to be amortized over a smaller number of larger updates.

A.6.2 ATOMIC OPERATIONS

Recall from Section 1.4 that mutexes can be used to ensure that multiple threads don't simultaneously try to update the same memory locations. However, modern CPUs and GPUs also provide specialized hardware instructions to perform certain operations *atomically*, generating consistent results when multiple threads use them to modify the same location concurrently. When applicable, atomics are generally more efficient than acquiring a mutex, updating the memory location, and releasing the mutex. Atomic instructions can only operate on a limited amount of memory (up to 8 bytes on current architectures) and support only a few operations (addition, swap, etc.). If atomic updates to more data or other kinds of operations are required, mutexes must generally be used instead.

C++11 provides a variety of atomic operations in the standard library, available via the `<atomic>` header file. For example, given the declaration of an integer value as `std::atomic` as follows, incrementing counter is an atomic operation.

```
std::atomic<int> counter(0);
:
:
counter++;
```

Atomic instructions do introduce some overhead, so they should only be used in cases where they are actually necessary.

Another useful atomic operation is “compare and swap,” which is also exposed by the C++ standard library. It takes a memory location and the value that the caller believes the location currently stores. If the memory location still holds that value when the atomic compare and swap executes, then a new value is stored and `true` is returned; otherwise, memory is left unchanged and `false` is returned.

Compare and swap is a building block that can be used to build many other atomic operations. For example, the code below could be executed by multiple threads to compute the maximum of values computed by all of the threads. (For this particular case, the specialized atomic maximum function would be a better choice, but this example helps convey the usage.)

```
std::atomic<int> maxValue;
int localMax = ...;
int currentMax = maxValue;
while (localMax > currentMax) {
    if (maxValue.compare_exchange_weak(currentMax, localMax))
        break;
}
```

If only a single thread is trying to update the memory location and the local value is larger, the loop is successful the first time through; the value loaded into `currentMax` is still the value stored by `maxValue` when `compare_exchange_weak()` executes and so `newMax` is

successfully stored and true is returned.⁴ If multiple threads are executing concurrently, then another thread may update the value in `maxValue` between the thread's read of `vaxValue` and the execution of `compare_exchange_weak()`. In that case, the compare and swap fails, memory isn't updated, and another pass is taken through the loop to try again. In the case of a failure, `compare_exchange_weak()` updates `currentMax` with the new value of `maxValue`.

An important application of atomic compare and swap is for the construction of data structures (as is done in Section 16.2.5 for photon mapping). Consider, for example, a tree data structure where each node has child node pointers initially set to `nullptr`. If code traversing the tree wants to create a new child at a node, code could be written like:

```
// atomic<Type *> node->firstChild
if (!node->firstChild) {
    Type *newChild = new Type ...
    Type *current = nullptr;
    if (node->firstChild.compare_exchange_weak(current, newChild) == false)
        delete newChild;
}
// node->firstChild != nullptr now
```

The idea is that if the child has the value `nullptr`, the thread speculatively creates and fully initializes the child node into a local variable, not yet visible to the other threads. Atomic compare and swap is then used to try to initialize the child pointer; if it still has the value `nullptr`, then the new child is stored and made available to all threads. If the child pointer no longer has the value `nullptr`, then another thread has initialized the child in the time between the current thread first seeing that it was `nullptr` and later trying to update it. In this case, the work done in the current thread turns out to have been wasted, but it can delete the locally created child node and continue execution, using the node created by the other thread.

This method of tree construction is a simple example of a *lock-free* algorithm. This approach has a few advantages compared to, for example, using a reader–writer mutex to manage updating the tree. First, there's no overhead of acquiring the reader mutex for regular tree traversal. Second, multiple threads can naturally concurrently update different parts of the tree. With a single reader–writer mutex, if one thread acquires the mutex to update one node in the tree, other threads won't be able to update other nodes. The “Further Reading” section at the end of the appendix has pointers to more information about lock-free algorithms.

A.6.3 ATOMIC FLOATING-POINT VALUES

The `std::atomic` template cannot be used with floating-point types. One of the main reasons that atomic operations are not supported with it is that floating-point operations are generally not commutative: as discussed in Section 3.9.1, when computed in floating-point, the value of the sum $(a+b)+c$ is not necessarily not equal to the sum $a+(b+c)$. In

⁴ The “weak” in the compare/exchange instruction refers to the shared memory model required of the underlying hardware. For our purposes, the lesser requirement of “weak” is fine, as it can be much more efficient than a strongly ordered memory model on some architectures. In return for this choice, the compare and exchange may occasionally fail incorrectly, so it requires a retry loop as we have implemented here.

turn, if a multi-threaded computation used atomic floating-point addition operations to compute some value, then the result computed wouldn't be the same across multiple program executions. (In contrast, with integer types, all of the supported operations are commutative, and so atomic operations give consistent results no matter which order threads perform them in.)

For pbrt's needs, these inconsistencies are generally tolerable, and being able to use atomic operations on `FLOAT`s is preferable in some cases to using a lock. (One example is splatting pixel contributions in the `Film::AddSplat()` method.) For these purposes, we provide a small `AtomicFloat` class.

```
(Parallel Declarations) ≡
class AtomicFloat {
public:
    (AtomicFloat Public Methods 1086)
private:
    (AtomicFloat Private Data 1086)
};
```

An `AtomicFloat` can be initialized from a provided floating-point value. In the implementation here, floating-point values are actually represented as their unsigned integer bitwise values, as returned by the `FloatToBits()` function.

```
(AtomicFloat Public Methods) ≡ 1086
explicit AtomicFloat(FLOAT v = 0) { bits = FloatToBits(v); }
```

By using a `uint32_t` to represent the value, we can use a `std::atomic` type to store it in memory, which in turn allows the compiler to be aware that the value in memory is being updated atomically. (If pbrt has been compiled to use 64-bit doubles for `FLOAT` values, a `uint64_t` is used instead, though this code isn't included here.)

```
(AtomicFloat Private Data) ≡ 1086
std::atomic<uint32_t> bits;
```

Assigning the value or returning it as a `FLOAT` is just a matter of converting to or from the unsigned integer representation.

```
(AtomicFloat Public Methods) +≡ 1086
operator FLOAT() const { return BitsToFloat(bits); }
FLOAT operator=(FLOAT v) { bits = FloatToBits(v); return v; }
```

Atomic floating-point addition is implemented via an atomic compare and exchange operation. In the do loop below, we convert the in-memory bit representation of the value to a `FLOAT`, add the provided difference in `v`, and attempt to atomically store the resulting bits. If the in-memory value has been changed by another thread since the value in `bits` was read from memory, the implementation continues retrying until the value in memory matches the expected value (in `oldBits`), at which point the atomic update succeeds.

AtomicFloat [1086](#)
 AtomicFloat::bits [1086](#)
 BitsToFloat() [212](#)
 Film::AddSplat() [494](#)
 FLOAT [1062](#)
 FloatToBits() [211](#)

```
(AtomicFloat Public Methods) +≡
    void Add(Float v) {
        uint32_t oldBits = bits, newBits;
        do {
            newBits = FloatToBits(BitsToFloat(oldBits) + v);
        } while (!bits.compare_exchange_weak(oldBits, newBits));
    }
```

1086

pbrt doesn't currently need to perform any other operations on `AtomicFloats`, so we don't provide any additional methods.

A.6.4 PARALLEL FOR LOOPS

All of the multi-core parallelism in pbrt is expressed through parallel for loops using the `ParallelFor()` function, which is implemented in the files `core/parallel.h` and `core/parallel.cpp`.⁵ `ParallelFor()` takes the loop body in the form of a function that is called for each loop iteration as well as a count of the total number of loop iterations to execute. It generally runs multiple iterations in parallel on different CPU cores and it returns only after all of the loop iterations have finished. In using `ParallelFor()`, the caller makes the implicit promise that it's safe to execute multiple loop iterations concurrently. An important implication of this promise is that the order in which the loop iterations are executed must not affect the final results computed.

Here is a simple example of using `ParallelFor()`. A C++ lambda expression is used to define the loop body; the loop index is passed back to it as an argument. The lambda has access to the local array variable and doubles each array element in its body. Note that the value 1024 passed as the second parameter to `ParallelFor()` after the lambda, giving the number of times to execute the loop body.

```
Float array[1024] = { ... };
ParallelFor(
    [array](int index) {
        array[index] *= 2.;
    }, 1024);
```

While it's also possible to pass a function pointer to `ParallelFor()`, lambdas are generally much more convenient given their ability to capture locally visible variables and make them available in their body.

For loops with relatively large iteration counts where the work done per iteration is small, it can be worthwhile to have the threads running loop iterations do multiple iterations before getting more work. (Doing so helps amortize the overhead of determining which iterations should be assigned to a thread.) Therefore, `ParallelFor()` also takes an optional `chunkSize` parameter that controls the granularity of the mapping of loop iterations to processing threads.

`AtomicFloat::bits` 1086

`BitsToFloat()` 212

`Float` 1062

`FloatToBits()` 211

⁵ Our implementation here, which is more efficient than the task system in the previous version of pbrt, is based on the parallel for loop implementation in *Halide* written by Jonathan Ragan-Kelley, Andrew Adams, and Zalman Stern.

(Parallel Definitions) ≡

```
void ParallelFor(const std::function<void(int)> &func,
    int count, int chunkSize) {
    (Run iterations immediately if not using threads or if count is small 1088)
    (Launch worker threads if needed 1088)
    (Create and enqueue ParallelForLoop for this loop 1089)
    (Notify worker threads of work to be done 1090)
    (Help out with parallel loop iterations in the current thread 1091)
}
```

ParallelFor() usually distributes loop iterations across multiple threads. However, if the system has only one CPU (or the user specified that only one thread should be used for rendering), or if the number of loop iterations is small, then the loop just runs immediately in the current thread, without any parallelism.

(Run iterations immediately if not using threads or if count is small) ≡ **1088**

```
if (PbrtOptions.nThreads == 1 || count < chunkSize) {
    for (int i = 0; i < count; ++i)
        func(i);
    return;
}
```

Parallel execution is implemented using a set of worker threads (a *thread pool*) that is created the first time ParallelFor() is called. The threads don't terminate after ParallelFor() returns, however; instead they wait on a condition variable that signals more work. This approach means that using the threads for parallel work is a fairly light-weight operation—the overhead of numerous operating system calls to create the threads is only paid once. (This implementation approach is often called *persistent threads*.) It's thus possible to use the thread pool for fairly fine-grained tasks, which in turn lets the system load-balance well when tasks have variable amounts of computation and lets the system scale well as more cores are available in the future.

(Parallel Local Definitions) ≡

```
static std::vector<std::thread> threads;
static bool shutdownThreads = false;
```

pbrt's initial execution thread also helps run loop iterations, so the number of worker threads launched is one fewer than the number of available CPU cores. There is thus a one-to-one relationship between cores and worker threads. Notwithstanding other processes running on the system, pbrt's threads are collectively enough to fully occupy the CPUs without introducing unnecessary thread-switching overhead from having more threads running than there are available cores. (NumSystemCores() returns the number of processing cores in the system.)

(Launch worker threads if needed) ≡ **1088, 1093**

```
if (threads.size() == 0) {
    ThreadIndex = 0;
    for (int i = 0; i < NumSystemCores() - 1; ++i)
        threads.push_back(std::thread(workerThreadFunc, i + 1));
}
```

NumSystemCores() **1088**
 Options::nThreads **1109**
 PbrtOptions **1109**
 ThreadIndex **1089**
 threads **1088**
 workerThreadFunc() **1092**

The function that worker threads run, `workerThreadFunc()`, will be introduced after we show how the state of enqueued parallel for loops is represented.

In the following, threads will need to determine which of the running threads they are. The `ThreadIndex` variable is declared with a qualifier that indicates that thread-local storage should be allocated for it, so that there is a separate instance of it for each thread. This variable is initialized to 0 for the main thread and goes from 1 to the number of threads for the worker threads.

```
(Parallel Declarations) +≡
    extern thread_local int ThreadIndex;
```

The `workList` variable holds a pointer to the head of a list of parallel for loops that aren't yet finished. Usually, there will be no more than one loop in this list, except in the presence of *nested parallelism*, when the body of one parallel for loop iteration specifies another parallel for loop in its body. The `workListMutex` must always be held when accessing `workList` or values stored in the `ParallelForLoop` objects held in it.

```
(Parallel Local Definitions) +≡
    static ParallelForLoop *workList = nullptr;
    static std::mutex workListMutex;
```

Adding a new loop to the work queue is fairly straightforward. After initializing the `ParallelForLoop` object that represents the loop's work, the implementation here locks the mutex and adds the loop to the head of the list. There are two important details here: first, because the call to `ParallelFor()` here doesn't return until all work for the loop is done, it's safe to allocate `loop` on the stack—no dynamic memory allocation is required.

Second, the loop is added to the *front* of the work list—doing so means that in the presence of nested parallelism, the inner loops will run before their enclosing loops. This leads to depth-first processing of the nested loops (rather than breadth-first), which in turn can avoid an explosion in the number of loops in the work list.

```
(Create and enqueue ParallelForLoop for this loop) ≡
    ParallelForLoop loop(func, count, chunkSize, CurrentProfilerState());
    workListMutex.lock();
    loop.next = workList;
    workList = &loop;
    workListMutex.unlock();

CurrentProfilerState() 1099
ParallelForLoop 1090
ParallelForLoop::next 1090
workList 1089
workListMutex 1089
```

The `ParallelForLoop` class encapsulates the relevant information about a parallel for loop body, including the function to run, the number of iterations, and which iterations are already done.

```
(Parallel Local Definitions) +≡
  class ParallelForLoop {
public:
  (ParallelForLoop Public Methods 1090)
public:
  (ParallelForLoop Private Data 1090)
  (ParallelForLoop Private Methods 1090)
};
```

ParallelForLoop can represent loops over both 1D and 2D domains corresponding to the variants of the ParallelFor() function. In the following, we'll only show the code for the 1D case.

```
(ParallelForLoop Public Methods) ≡ 1090
  ParallelForLoop(std::function<void(int)> func1D,
                  int64_t maxIndex, int chunkSize, int profilerState)
    : func1D(std::move(func1D)), maxIndex(maxIndex),
      chunkSize(chunkSize), profilerState(profilerState) { }
```

```
(ParallelForLoop Private Data) ≡ 1090
  std::function<void(int)> func1D;
  const int64_t maxIndex;
  const int chunkSize, profilerState;
```

The nextIndex member variable tracks the next loop index to be executed. It is incremented by workers as they claim loop iterations to execute in their threads. The value stored in activeWorkers records how many worker threads are currently running iterations of the loop. next is used to maintain the linked list of nested loops.

```
(ParallelForLoop Private Data) +≡ 1090
  int64_t nextIndex = 0;
  int activeWorkers = 0;
  ParallelForLoop *next = nullptr;
```

A parallel for loop is only finished when the index has been advanced to the end of the loop's range and there are no threads currently working on it. Note that the first of these conditions will be reached while work is still in progress.

```
(ParallelForLoop Private Methods) ≡ 1090
  bool Finished() const { return nextIndex >= maxIndex &&
                           activeWorkers == 0; }
```

After the loop has been added to the work list, the worker threads are signaled so that they wake up and start taking work from the list.

```
(Notify worker threads of work to be done) ≡ 1088
  std::unique_lock<std::mutex> lock(workListMutex);
  workListCondition.notify_all();
```

```
(Parallel Local Definitions) +≡
  static std::condition_variable workListCondition;
```

```
ParallelFor() 1088
ParallelForLoop 1090
ParallelForLoop::
  activeWorkers
  1090
ParallelForLoop::chunkSize
  1090
ParallelForLoop::func1D 1090
ParallelForLoop::maxIndex
  1090
ParallelForLoop::nextIndex
  1090
ParallelForLoop::
  profilerState
  1090
workListCondition 1090
workListMutex 1088
```

Finally, the thread that called `ParallelFor()` (be it the main thread or one of the worker threads) starts work on the loop. In the presence of nested parallelism, this means that the thread that enqueued this loop works on it exclusively before returning. By finishing the loop before allowing the thread that submitted it to do any more work, the implementation keeps the amount of enqueued work limited and allows subsequent code in the caller to proceed, knowing the loop's work is done after its call to `ParallelFor()` returns.

A lock to `workListMutex` is always held going into the `while` loop here. Note that the lock is necessary even for calling the `Finished()` method, since `loop` is stored in `workList` and thus will be accessed by the other threads.

(Help out with parallel loop iterations in the current thread) ≡
 while (!loop.Finished()) {
(Run a chunk of loop iterations for loop 1091)
 }

1088

Each time through the `while` loop, the thread runs one or more iterations of the parallel loop's body.

(Run a chunk of loop iterations for loop) ≡
(Find the set of loop iterations to run next 1091)
(Update loop to reflect iterations this thread will run 1091)
(Run loop indices in [indexStart, indexEnd) 1092)
(Update loop to reflect completion of iterations 1092)

1091, 1093

The range of iterations goes from the current index to `chunkSize` ahead subject to the total number of iterations.

(Find the set of loop iterations to run next) ≡
 int64_t indexStart = loop.nextIndex;
 int64_t indexEnd = std::min(indexStart + loop.chunkSize, loop.maxIndex);

1091

Now that the thread has found the iterations that it will run, `loop` must be updated. If this thread took the final iterations, the loop is removed from the work list so that other threads can start on the next loop (if any).

```
ParallelForLoop::  

activeWorkers  

1090  

ParallelForLoop::chunkSize  

1090  

ParallelForLoop::Finished()  

1090  

ParallelForLoop::maxIndex  

1090  

ParallelForLoop::next 1090  

ParallelForLoop::nextIndex  

1090  

workList 1089  

workListMutex 1089
```

1091

Given the range of loop iterations to run, it's fairly straightforward to call back to the `std::function` representing the loop body. This is the only time in the enclosing `while` loop that the lock is relinquished, though the time spent running these loop iterations is generally the majority of the time spent in the `while` loop, so other worker threads generally don't need to wait long for the lock. The *(Handle other types of loops)* fragment, not included here, handles the 2D loop supported by `ParallelForLoop`.

```
{Run loop indices in [indexStart, indexEnd]} ≡ 1091
lock.unlock();
for (int index = indexStart; index < indexEnd; ++index) {
    if (loop.func1D) {
        loop.func1D(index);
    }
    {Handle other types of loops}
}
lock.lock();
```

After running the set of loop iterations and re-acquiring the lock, the active worker count is updated to reflect that (for now at least) the current thread is no longer working on loop.

```
{Update loop to reflect completion of iterations} ≡ 1091
loop.activeWorkers--;
```

While the thread that called `ParallelFor()` is working on the loop, the other threads also run iterations. `workerThreadFunc()` is the function that runs to do this in each task execution thread. Its structure is similar to the fragment *{Help out with parallel loop iterations in the main thread}*, with three main differences. First, it runs loops from whatever `ParallelForLoops` are in `workList`, not just from a single parallel for loop. Second, it has the thread sleep whenever there isn't any work to be done. Finally, it continues waiting for more loops to run until the `shutdownThreads` variable is set, which only happens at the end of program execution.

As before, a lock to the `workListMutex` must be held at entry to the `while` loop here.

```
{Parallel Local Definitions} +≡
static void workerThreadFunc(int tIndex) {
    ThreadIndex = tIndex;
    std::unique_lock<std::mutex> lock(workListMutex);
    while (!shutdownThreads) {
        if (!workList) {
            {Sleep until there are more tasks to run 1092}
        } else {
            {Get work from workList and run loop iterations 1093}
        }
    }
    {Report thread statistics at worker thread exit 1093}
}
```

If there is no available work, the worker thread waits on the `workListCondition` condition variable. The semantics of condition variables are such that doing so releases the lock, but when this thread is later woken up by the condition variable being signaled, it will again hold the lock.

```
{Sleep until there are more tasks to run} ≡ 1092
workListCondition.wait(lock);
```

```
ParallelForLoop::
activeWorkers 1090
ParallelForLoop::func1D 1080
shutdownThreads 1088
ThreadIndex 1089
workList 1089
workListCondition 1090
workListMutex 1089
```

Otherwise, a range of loop iterations to run is taken from the head of `workList`. The code to run iterations is reused from the *(Run a chunk of loop iterations for loop)* fragment defined earlier.

```
<Get work from workList and run loop iterations> ≡ 1092
    ParallelForLoop &loop = *workList;
    (Run a chunk of loop iterations for loop 1091)
    if (loop.Finished())
        workListCondition.notify_all();
```

Finally, as will be discussed shortly in Appendix A.7.1, the worker thread must call `ReportThreadStats()` before exiting so that its per-thread statistics are merged into the aggregate statistics.

```
<Report thread statistics at worker thread exit> ≡ 1092
    ReportThreadStats();
```

A variant of `ParallelFor()` takes a `Point2i` to describe a 2D iteration domain from $(0, 0)$ to the given point. This version is used to loop over image buckets in Section 1.3.4, for example.

```
<Parallel Declarations> +≡
void ParallelFor2D(std::function<void(Point2i)> func,
                    const Point2i &count);
```

The `ThreadIndex` variable allows code in parallel for loops to use preallocated temporary buffers or objects like `MemoryArenas`, giving a separate instance to each worker thread without needing to worry about data races. (See, for example, the *(Generate SPPM visible points)* fragment on page 976.) For this use, it's also useful for calling code to be able to find out the maximum possible thread index.

```
<Parallel Definitions> +≡
int MaxThreadIndex() {
    if (PbrtOptions.nThreads != 1) {
        (Launch worker threads if needed 1088)
    }
    return 1 + threads.size();
}
```

MemoryArena 1074
 Options::nThreads 1109
 ParallelForLoop 1090
`ParallelForLoop::Finished()` 1090
 PbrtOptions 1109
`Point2i` 68
`ReportThreadStats()` 1097
 ThreadIndex 1089
`threads` 1088
`workList` 1089
`workListCondition` 1090

A.7 STATISTICS

Collecting data about the run-time behavior of the system can provide a substantial amount of insight into its behavior and opportunities for improving its performance. For example, we might want to track the average number of primitive intersection tests performed for all of the rays; if this number is surprisingly high, then there may be a latent bug somewhere in the system. `pbrt`'s statistics system makes it possible to measure and aggregate this sort of data in a variety of ways.

It's important to make it as easy as possible to add new measurements to track the system's run-time behavior; the easier it is to do this, the more measurements end up being added to the system, and the more likely that "interesting" data will be discovered, leading to new insights and improvements. Therefore, adding new measurements to the system is fairly straightforward. For example, the following lines declare two counters that can be used to record how many times the corresponding events happen.

```
STAT_COUNTER("Integrator/Regular ray intersection tests",
             nIntersectionTests);
STAT_COUNTER("Integrator/Shadow ray intersection tests",
             nShadowTests);
```

As appropriate, counters can be incremented with simple statements like

```
++nIntersectionTests;
```

With no further intervention from the developer, the preceding is enough for the statistics system to be able to automatically print out nicely formatted results like the following when rendering completes:

Integrator	
Regular ray intersection tests	752982
Shadow ray intersection tests	4237165

The statistics system supports the following measurements:

- **STAT_COUNTER("name", var):** A simple count of the number of instances of an event. The counter variable `var` can be updated as if it was a regular `int` variable; for example, `++var` and `var += 10` are both valid.
- **STAT_MEMORY_COUNTER("name", var):** A specialized counter for recording memory usage. In particular, the values reported at the end of rendering are in terms of kilobytes, megabytes, or gigabytes, as appropriate. The counter is updated the same way as a regular counter: `var += count * sizeof(MyStruct)` and so forth.
- **STAT_INT_DISTRIBUTION("name", dist):** Tracks the distribution of some value; at the end of rendering, the minimum, maximum, and average of the reported values are reported. Call `ReportValue(dist, val)` to include `val` in the distribution.
- **STAT_FLOAT_DISTRIBUTION("name", dist):** This counter also tracks the distribution of a value but expects floating-point values to be reported. `ReportValue(dist, val)` is also used to report values.
- **STAT_PERCENT("name", num, denom):** Tracks how often a given event happens; the aggregate value is reported as the percentage `num/denom` when statistics are printed. Both `num` and `denom` can be incremented as if they were integers—for example, one might write `if (event) ++num; or ++denom.`
- **STAT_RATIO("name", num, denom):** This tracks how often an event happens but reports the result as a ratio `num/denom` rather than a percentage. This is often a more useful presentation if `num` is often greater than `denom`. (For example, we might record the percentage of ray-triangle intersection tests that resulted in an intersection but the ratio of triangle intersection tests to the total number of rays traced.)
- **STAT_TIMER("name", timer):** The timer can be used to record how much time is spent in a given block of code. To run the timer, one can add a declaration such as `StatTimer t(&timer);` to the start of a block of code; the timer will run until the

declared `t` variable goes out of scope and the `StatTimer` destructor runs. Note that there is some overhead to getting the current system time, so timers shouldn't be used for very small blocks of code.

All of the macros to define statistics trackers can only be used at file scope and should only be used in .cpp files (for reasons that will become apparent as we dig into their implementations). They specifically should not be used in header files or function or class definitions.

Note also that the string names provided for each measurement should be of the form “category/statistic.” When values are reported, everything under the same category is reported together (as in the preceding example).

A.7.1 IMPLEMENTATION

There are a number of challenges in making the statistics system both efficient and easy to use. The efficiency challenges stem from pbrt being multi-threaded: if there wasn't any parallelism, we could associate regular integer or floating-point variables with each measurement and just update them like regular variables. In the presence of multiple concurrent threads of execution, however, we need to ensure that two threads didn't try to modify these variables at the same time (recall the discussion of mutual exclusion in Section 1.4).

While atomic operations like those described in Section A.6.2 could be used to safely increment counters without using a mutex, there would still be a performance impact from multiple threads modifying the same location in memory. Recall from Section A.6.1 that the cache coherence protocols can introduce substantial overhead in this case. Because the statistics measurements are updated so frequently during the course of rendering, we found that an atomics-based implementation caused the overall renderer to be 10–15% slower than the current implementation, which avoids the overhead of multiple threads frequently modifying the same memory location.

The implementation here is based on having separate counters for each running thread, allowing the counters to be updated without atomics and without cache coherence overhead (since each thread increments its own counters). This approach means that in order to report statistics, it's necessary to merge all of these per-thread counters into final aggregate values, which we'll see is possible with a bit of trickiness.

To see how this all works, we'll dig into the implementation for regular counters; the other types of measurements are all along similar lines. First, here is the `STAT_COUNTER` macro, which packs three different things into its definition.

(Statistics Macros) ≡

```
StatRegisterer 1096
StatsAccumulator 1096
StatsAccumulator:::
ReportCounter()
1096
#define STAT_COUNTER(title, var) \
    static thread_local int64_t var; \
    static void STATS_FUNC##var(StatsAccumulator &accum) { \
        accum.ReportCounter(title, var); \
        var = 0; \
    } \
    static StatRegisterer STATS_REG##var(STATS_FUNC##var)
```

First, and most obviously, the macro defines a 64-bit integer variable named `var`, the second argument passed to the macro. The variable definition has the `thread_local` qualifier, which indicates that there should be a separate copy of the variable for each executing thread. Given these per-thread instances, we need to be able to sum together the per-thread values and to aggregate all of the individual counters into the final program output.

To this end, the macro next defines a function, giving it a (we hope!) unique name derived from `var`. When called, this function passes along the value of `var` for the current thread to an instance of the `StatsAccumulator` class. `StatsAccumulator` accumulates values of measurement statistics counters into its own storage; here is its `ReportCounter()` method, which totals up the given values, associating them with their respective string names.

```
(StatsAccumulator Public Methods) ≡
void ReportCounter(const std::string &name, int64_t val) {
    counters[name] += val;
}

(StatsAccumulator Private Data) ≡
std::map<std::string, int64_t> counters;
```

All we need now is for the `STATS_FUNC##var()` function to be called for each per-thread instance of each counter variable when rendering has finished; this is what the last line of the `STAT_COUNTER` macro takes care of. To understand what it does, first recall that in C++, constructors of global static objects run when program execution starts; thus, each static instance of the `StatRegisterer` class runs its constructor before `main()` starts running.

The constructor for `StatRegisterer` objects in turn adds the function passed to it to a vector that holds all of the various `STATS_FUNC##var()` functions. Note that because the `StatRegisterer` constructors run at program start time, the constructor for the `funcs` vector (another global static) may not yet have run. Therefore, `funcs` is a pointer to a vector rather than a vector, so that it can be explicitly constructed before it's first used.

```
(StatRegisterer Public Methods) ≡
StatRegisterer(std::function<void(StatsAccumulator &)> func) {
    if (!funcs)
        funcs = new std::vector<std::function<void(StatsAccumulator &)>>;
    funcs->push_back(func);
}

(StatRegisterer Private Data) ≡
static std::vector<std::function<void(StatsAccumulator &)>> *funcs;
```

Now all the pieces can come together. Each thread launched for multi-threading in `pbrt` runs the function `workerThreadFunc()`. When this function exits at the end of program execution, it calls `ReportThreadStats()`, which in turn causes the per-thread measurements for the thread to be merged into the given `StatsAccumulator`. (The initial processing thread also calls this function after rendering has completed so that its values are reported.)

The `ReportThreadStats()` method uses a mutex to ensure that no other thread can be updating the `StatsAccumulator` concurrently and then dispatches to `StatRegisterer::CallCallbacks()`, passing in a single `StatsAccumulator` to store all of the aggregated values.

```
(Statistics Definitions) ≡
void ReportThreadStats() {
    static std::mutex mutex;
    std::lock_guard<std::mutex> lock(mutex);
    StatRegisterer::CallCallbacks(statsAccumulator);
}
```

```
(Statistics Local Variables) ≡
static StatsAccumulator statsAccumulator;
```

Finally, `CallCallbacks()` calls each of the function pointers to the `STATS_FUNC##var()` functions that were assembled when the program started. This is where the magic happens: because these functions are called repeatedly, once in each of the threads, each call to the `STATS_FUNC##var()` functions reports the value of the per-thread value for each thread. In the end, we will have aggregated all of the various measurement values.

```
(Statistics Definitions) +≡
void StatRegisterer::CallCallbacks(StatsAccumulator &accum) {
    for (auto func : *funcs)
        func(accum);
}
```

The `PrintStats()` function prints all of the statistics that have been accumulated in `StatsAccumulator`. This function is called by `pbrtWorldEnd()` at the end of rendering. Its task is straightforward: it sorts the measurements by category and formats the values so that columns of numbers line up and so forth.

A.7.2 PROFILING

When doing significant work on optimizing the performance of any sort of software system, using a stand-alone profiler is a necessity. Modern profilers provide a wealth of information about program execution—which functions, and even which lines of code most execution time is spent in. Equally useful, many also provide information about the program’s interaction with the computer system—which lines of code suffered the most from CPU cache misses, places in the program where cache coherence overhead was high, and so forth.

We have found that a useful supplement to stand-alone profilers is to have a simple profiling system embedded in `pbrt`.⁶ This system provides high-level information about the fraction of overall run time consumed by various parts of the system. The information it provides is useful for understanding the differences across scenes where `pbrt` spends its time and for helping to guide more focused profiling investigations. In exchange for

```
pbrtWorldEnd() 1129
StatRegisterer::
    CallCallbacks()
    1097
StatRegisterer::funcs 1096
StatsAccumulator 1096
statsAccumulator 1097
```

⁶ The design follows the approach used in Disney’s Hyperion renderer, as described by Selle (2015).

only providing high-level information, it adds minimal overhead; in practice, we have measured a roughly 2.5% increase in run time.

Here is an example of the profiler's output for one of the test scenes:

Integrator::Render()	95.0 %
Sampler::StartPixelSample()	10.1 %
SamplerIntegrator::Li()	81.1 %
Accelerator::Intersect()	3.6 %
Triangle::Intersect()	1.1 %
BSDF::f()	0.1 %
Direct lighting	75.1 %
Accelerator::Intersect()	16.4 %
Triangle::Intersect()	4.3 %

There are a few things to know in order to understand this output. First, all percentages reported are with respect to the system's overall run time. Thus, here we can see that about 5% of the run time was spent parsing the scene file and constructing the scene, and the remainder was spent rendering it. Most of the rendering time was spent doing direct lighting calculations.

Second, note that there is a hierarchy in the reported results. For example, Accelerator::Intersect() appears twice, and we can see that 3.6% of the run time was spent in calls to the accelerator's traversal method from Li(), but 16.4% of the run time was spent in calls to it from the direct lighting code. The percentage reported for each level of this hierarchy includes the time of all of the indented items reported underneath it.

The profiling system is based on explicit annotations in the program code to denote these phases of execution. (We haven't included those annotations in the program source code in the book in the interests of simplicity.) In turn, pbrt uses operating-system-specific functionality to periodically interrupt the system and record which phases are currently active. At the end of execution, it counts up how many times each phase was found to be active and reports these counts with respect to the totals, as above.

As suggested by the profiling output, there are a series of categories that a phase of execution can be charged to; these are represented by the Prof enumerant.

```
(Statistics Declarations) ≡
enum class Prof {
    IntegratorRender,     SamplerIntegratorLi,     DirectLighting,
    AccelIntersect,        AccelIntersectP,          TriIntersect,
    (Remainder of Prof enum entries)
};
```

The hierarchy of reported percentages is implicitly defined by the ordering of values in this enumerant; when multiple execution phases are active, later values in the enumerant are assumed to be children of earlier ones.

One bit in the ProfilerState variable is used to indicate whether each category in Prof is currently active. Note that this variable has the `thread_local` qualifier; thus, a separate instance is maintained for each program execution thread.

```
{Statistics Declarations} +≡
    extern thread_local uint32_t ProfilerState;
```

`CurrentProfilerState()` makes the state available to the `ParallelFor()` implementation; this makes it possible for work done in parallel for loop bodies to be charged to the execution phases that were active when `ParallelFor()` was called.

```
{Statistics Declarations} +≡
    inline uint32_t CurrentProfilerState() { return ProfilerState; }
```

In turn, a method like `BVHAccel::Intersect()` starts with a single line of code for the profiling system's benefit:

```
ProfilePhase p(Prof::AccelIntersect);
```

The `ProfilePhase` helper class's constructor records that the provided phase of execution has begun; when its destructor later runs when the `ProfilePhase` instance goes out of scope, it records that the corresponding phase is no longer active.

```
{Statistics Declarations} +≡
class ProfilePhase {
public:
    <ProfilePhase Public Methods 1099>
private:
    <ProfilePhase Private Data 1099>
};
```

The constructor starts by mapping the provided phase to a bit and then enabling that bit in `ProfilerState`. In the presence of recursion (notably, via recursive calls to `SamplerIntegrator::Li()`), it's possible that a higher level of the recursion will already have indicated the start of an execution phase. In this case, `reset` is set to `false`, and the `ProfilePhase` destructor won't clear the corresponding bit when it runs, leaving that task to the destructor of the earlier `ProfilePhase` instance that first set that bit.

```
<ProfilePhase Public Methods> ≡
ProfilePhase(Prof p) {
    categoryBit = (1 << (int)p);
    reset = (ProfilerState & categoryBit) == 0;
    ProfilerState |= categoryBit;
}
```

`BVHAccel::Intersect()` 282

`ParallelFor()` 1088

`Prof` 1098

`ProfilePhase` 1099

`ProfilePhase::categoryBit` 1099

`ProfilePhase::reset` 1099

`ProfilerState` 1099

`SamplerIntegrator::Li()` 31

```
<ProfilePhase Private Data> ≡
bool reset;
uint32_t categoryBit;
```

```
<ProfilePhase Public Methods> +≡
~ProfilePhase() {
    if (reset)
        ProfilerState &= ~categoryBit;
}
```

1099

1099

1099

The `InitProfiler()` function has two tasks. First, it allocates the `profileSamples` array; this array has one entry for each possible combination of active phases of execution. While many of these combinations aren't actually possible, using a simple data structure like this makes it possible to very efficiently record the sample counts of the current phase of execution. (However, note that the number of entries grows at the rate 2^n , where n is the number of distinct phases of execution in the `Prof` enumerant. Thus, this implementation is not suited to extremely fine-grained decomposition of profiling categories.)

Second, `InitProfiler()` uses system-dependent functionality to ask that `pbrt` be periodically interrupted to record the current phase of execution. We won't include this code here.

(Statistics Definitions) +≡

```
void InitProfiler() {
    profileSamples.reset(new std::atomic<uint64_t>[1 << NumProfEvents]);
    for (int i = 0; i < (1 << NumProfEvents); ++i)
        profileSamples[i] = 0;
    {Set timer to periodically interrupt the system for profiling}
}
```

(Statistics Local Variables) +≡

```
static std::unique_ptr<std::atomic<uint64_t>[]> profileSamples;
```

Each time the system interrupt occurs for a thread, `ReportProfileSample()` is called. This function is extremely simple and just increments the tally for the current set of active phases as encoded in `ProfilerState`.

(Statistics Definitions) +≡

```
static void ReportProfileSample(int, siginfo_t *, void *) {
    if (profileSamples)
        profileSamples[ProfilerState]++;
}
```

At the end of rendering, the `ReportProfilerResults()` function is called. It aggregates the profile counters and generates reports of the form above; its implementation isn't particularly interesting with respect to its length, so we won't include it here.

FURTHER READING

Hacker's Delight (Warren 2006), is a delightful and thought-provoking exploration of the bit-twiddling algorithms like those used in some of the utility routines in this appendix. Sean Anderson (2004) has a Web page filled with a collection of bit-twiddling techniques like the ones in `IsPowerOf2()` and `RoundUpPow2()` at graphics.stanford.edu/~seander/bithacks.html.

Numerical Recipes, by Press et al. (1992), and Atkinson's book (1993) on numerical analysis both discuss algorithms for matrix inversion and solving linear systems.

The PCG random number generator was developed by O'Neill (2014). The paper describing its implementation is well written and also features extensive discussion of

`IsPowerOf2()` 1064

`Prof` 1098

`ProfilerState` 1099

`profileSamples` 1100

`RoundUpPow2()` 1064

a range of previous pseudo-random number generators and the challenges that they have faced in passing rigorous tests of their quality (L'Ecuyer and Simard 2007).

Many papers have been written on cache-friendly programming techniques; only a few are surveyed here. Ericson's chapter (2004) on high-performance programming techniques has very good coverage of this topic. Lam, Rothberg, and Wolf (1991) investigated blocking (tiling) for improving cache performance and developed techniques for selecting appropriate block sizes, given the size of the arrays and the cache size. Grunwald, Zorn, and Henderson (1993) were one of the first groups of researchers to investigate the interplay between memory allocation algorithms and the cache behavior of applications.

In `pbrt`, we only worry about cache layout issues for dynamically allocated data. However, Calder et al. (1998) show a profile-driven system that optimizes memory layout of global variables, constant values, data on the stack, and dynamically allocated data from the heap in order to reduce cache conflicts among them all, giving an average 30% reduction in data cache misses for the applications they studied.

Blocking for tree data structures was investigated by Chilimbi et al. (1999b); they ensured that nodes of the tree and a few levels of its children were allocated contiguously. Among other applications, they applied their tool to the layout of the acceleration octree in the *Radiance* renderer and reported a 42% speedup in run time. Chilimbi et al. (1999a) also evaluated the effectiveness of reordering fields inside structures to improve locality.

Drepper's paper (2007) is a useful resource for understanding performance issues related to caches, cache coherence, and main memory access, particularly in multicore systems.

Boehm's paper *Threads Cannot Be Implemented as a Library* (2005) makes the remarkable (and disconcerting) observation that multi-threading cannot be reliably implemented without the compiler having explicit knowledge of the fact that multi-threaded execution is expected. Boehm presented a number of examples that demonstrate the corresponding dangers in 2005-era compilers and language standards like C and C++ that did not have awareness of threading. Fortunately, the C++11 and C11 standards addressed the issues that he identified.

`pbrt`'s parallel `for` loop-based approach to multi-threading is a widely used technique for multi-threaded programming; the OpenMP standard supports a similar construct (and much more) (OpenMP Architecture Review Board 2013). A slightly more general model for multi-core parallelism is available from *task systems*, where computations are broken up into a set of independent tasks that can be executed concurrently. Blumofe et al. (1996) described the task scheduler in Cilk, and Blumofe and Leiserson (1999) describe the work-stealing algorithm that is the mainstay of many current high-performance task systems.

EXERCISES

- ➊ A.1 Modify `MemoryArena` so that it just calls `new` for each memory allocation. (You will also want to record all pointer values returned by these `new` calls so that memory can be freed in the `Reset()` method.) Render images of a few scenes and measure how much more slowly `pbrt` runs. Can you quantify how much of

this is due to different cache behavior and how much is due to overhead in the dynamic memory management routines?

- ➊ A.2 Change the `BlockedArray` class so that it doesn't do any blocking and just uses a linear addressing scheme for the array. Measure the change in pbrt's performance as a result. (Scenes with many image map textures are most likely to show any differences, since the `MIPMap` class is a key user of `BlockedArray`.)
- ➋ A.3 Try a few alternative implementations of the statistics system described in Section A.7 to get a sense of the performance trade-offs with various approaches. You might try using atomic operations to update single counters that are shared across threads, or you might try using a mutex to allow safe updates to shared counters by multiple threads. Measure the performance compared to pbrt's current implementation and discuss possible explanations for your results.

SCENE DESCRIPTION INTERFACE

This appendix describes the application programming interface (API) that is used to describe the scene to be rendered to `pbrt`. Users of the renderer typically don't call the functions in this interface directly but instead describe their scenes using the text file format described in documentation on the `pbrt` Web site (`pbrt.org`). The statements in these text files have a direct correspondence to the API functions described here.

The need for such an interface to the renderer is clear: there must be a convenient way in which all of the properties of the scene to be rendered can be communicated to the renderer. The interface should be well defined and general purpose, so that future extensions to the system fit into its structure cleanly. It shouldn't be too complicated, so that it's easy to describe scenes, but it should be expressive enough that it doesn't leave any of the renderer's capabilities hidden.

A key decision to make when designing a rendering API is whether to expose the system's internal algorithms and structures or offer a high-level abstraction for describing the scene. These have historically been the two main approaches to scene description in graphics: the interface may specify *how* to render the scene, configuring a rendering pipeline at a low level using deep knowledge of the renderer's internal algorithms, or it may specify *what* the scene's objects, lights, and material properties are and leave it to the renderer to decide how to transform that description into the best possible image.

The first approach has been successfully used for interactive graphics. In APIs such as OpenGL® or Direct3D®, it is not possible to just mark an object as a mirror and have reflections appear automatically; rather, the user must choose an algorithm for rendering reflections, render the scene multiple times (e.g., to generate an environment map), store those images in a texture, and then configure the graphics pipeline to use the environment map when rendering the reflective object. The advantage of this approach

is that the full flexibility of the rendering pipeline is exposed to the user, making it possible to carefully control the actual computation being done and to use the pipeline very efficiently. Furthermore, because APIs like these impose a very thin abstraction layer between the user and the renderer, the user can be confident that unexpected inefficiencies won't be introduced by the API.

The second approach to scene description, based on describing the geometry, materials, and lights at a higher level of abstraction, has been most successful for applications like high-quality offline rendering. There, users are generally willing to cede control of the low-level rendering details to the renderer in exchange for the ability to specify the scene's properties at a high level. An important advantage of the high-level approach is that the implementations of these renderers have greater freedom to make major changes to the internal algorithms of the system, since the API exposes less of them.

For `pbrt`, we will use an interface based on the descriptive approach. Because `pbrt` is fundamentally physically based, the API is necessarily less flexible in some ways than APIs for many nonphysically based rendering packages. For example, it is not possible to have some lights illuminate only some objects in the scene.

Another key decision to make in graphics API design is whether to use an immediate mode or a retained mode style. In an immediate mode API, the user specifies the scene via a stream of commands that the renderer processes as they arrive. In general, the user cannot make changes to the scene description data already specified (e.g., “change the material of that sphere I described previously from plastic to glass”); once it has been given to the renderer, the information is no longer accessible to the user. Retained mode APIs give the user some degree of access to the data structures that the renderer has built to represent the scene. The user can then modify the scene description in a variety of ways before finally instructing the renderer to render the scene.

Immediate mode has been very successful for interactive graphics APIs since it allows graphics hardware to draw the objects in the scene as they are supplied by the user. Since they do not need to build data structures to store the scene and since they can apply techniques like immediately culling objects that are outside of the viewing frustum without worrying that the user will change the camera position before rendering, these APIs have been key to high-performance interactive graphics.

For ray-tracing-based renderers like `pbrt`, where the entire scene must be described and stored in memory before rendering can begin, some of these advantages of an immediate mode interface aren't applicable. Nonetheless, we will use immediate mode semantics in our API, since it leads to a clean and straightforward scene description language. This choice makes it more difficult to use `pbrt` for applications like quickly rerendering a scene after making a small change to it (e.g., by moving a light source) and may make rendering animations less straightforward, since the entire scene needs to be redescribed for each frame of an animation. Adding a retained mode interface to `pbrt` would be a challenging but useful project.

`pbrt`'s rendering API consists of just over 40 carefully chosen functions, all of which are declared in the `core/api.h` header file. The implementation of these functions is in `core/api.cpp`. This appendix will focus on the general process of turning the API function calls into instances of the classes that represent scenes.

B.1 PARAMETER SETS

A key problem that a rendering API must address is extensibility—as new features are added to the system, how does the user-visible API change and what parts of its implementation change? For pbrt, it’s important that developers be able to easily add new implementations of Shapes, Cameras, Integrators, and so forth. We’ve designed the API with this goal in mind.

To this end, the caller-visible API and its implementation are both as unaware as possible of what particular parameters these objects take and what their semantics are. pbrt uses the `ParamSet` class to bundle up parameters and their values in a generic way. For example, it might record that there is a single floating-point value named “radius” with a value of 2.5 and an array of four color values named “specular” with various SPDs. The `ParamSet` provides methods for both setting and retrieving values from these kinds of generic parameter lists. It is defined in `core/paramset.h` and `core/paramset.cpp`.

Most of pbrt’s API routines take a `ParamSet` as one of their parameters; for example, the shape creation routine, `pbrtShape()`, just takes a string giving the name of the shape to make and a `ParamSet` with parameters for it. The creation routine of the corresponding shape implementation is called with the `ParamSet` passed along as a parameter; it extracts values from the `ParamSet` to get parameters to use in a call to the class’s constructor.

```
<ParamSet Declarations> ≡
class ParamSet {
public:
    <ParamSet Public Methods 1107>
private:
    <ParamSet Private Data 1105>
};
```

A `ParamSet` can hold eleven types of parameters: Booleans, integers, floating-point values, points (2D and 3D), vectors (2D and 3D), normals, spectra, strings, and the names of Textures that are being used as parameters for Materials and other Textures. Internally, it stores a vector of named values for each of the different types that it stores; each parameter is represented by a pointer to a `ParamsetItem` of the appropriate type. A `shared_ptr` is used for these pointers; doing so allows a parameter to easily be stored in multiple `ParamSets`, which we’ll find useful in the following.

Float 1062
Integrator 25
Material 577
Normal3f 71
ParamSet 1105
ParamsetItem 1106
pbrtShape() 1124
Point2f 68
Point3f 68
Spectrum 315
Texture 614
Vector2f 60
Vector3f 60

Storing parameters unsorted in vectors means that searching for a given parameter takes $O(n)$ time, where n is the number of parameters of the parameter’s type. In practice, there are just a handful of parameters to any function, so a more time-efficient representation isn’t necessary.

```
<ParamSet Private Data> ≡
std::vector<std::shared_ptr<ParamsetItem<bool>>> bools;
std::vector<std::shared_ptr<ParamsetItem<int>>> ints;
std::vector<std::shared_ptr<ParamsetItem<Float>>> floats;
std::vector<std::shared_ptr<ParamsetItem<Point2f>>> point2fs;
std::vector<std::shared_ptr<ParamsetItem<Vector2f>>> vector2fs;
std::vector<std::shared_ptr<ParamsetItem<Point3f>>> point3fs;
```

1105

```
std::vector<std::shared_ptr<ParamSetItem<Vector3f>>> vector3fs;
std::vector<std::shared_ptr<ParamSetItem<Normal3f>>> normals;
std::vector<std::shared_ptr<ParamSetItem<Spectrum>>> spectra;
std::vector<std::shared_ptr<ParamSetItem<std::string>>> strings;
std::vector<std::shared_ptr<ParamSetItem<std::string>>> textures;
```

B.1.1 THE ParamSetItem STRUCTURE

The ParamSetItem structure stores all of the relevant information about a single parameter, such as its name, its base type, and its value(s). For example (using the syntax from pbrt's input files), the `foo` parameter

```
"float foo" [ 0 1 2 3 4 5 ]
```

has a base type of `float`, and six values have been supplied for it. It would be represented by a `ParamSetItem<Float>`.

```
(ParamSet Declarations) +≡
template <typename T> struct ParamSetItem {
    (ParamSetItem Public Methods)
    (ParamSetItem Data 1106)
};
```

The `ParamSetItem` directly initializes its members from the arguments and makes a copy of the values.

```
(ParamSetItem Methods) ≡
template <typename T>
ParamSetItem<T>::ParamSetItem(const std::string &name, const T *v,
                             int nValues)
: name(name), values(new T[nValues]), nValues(nValues) {
    std::copy(v, v + nValues, values.get());
}
```

The Boolean value `lookedUp` is set to `true` after the value has been retrieved from the `ParamSet`. This makes it possible to print warning messages if any parameters were added to the parameter set but never used, which typically indicates a misspelling in the scene description file or other user error.

```
(ParamSetItem Data) ≡
const std::string name;
const std::unique_ptr<T[]> values;
const int nValues;
mutable bool lookedUp = false;
```

1106

B.1.2 ADDING TO THE PARAMETER SET

To add an entry to the parameter set, the appropriate `ParamSet` method should be called with the name of the parameter, a pointer to its data, and the number of data items. These methods first remove previous values for the parameter in the `ParamSet`, if any.

ParamSetItem 1106
ParamSetItem::name 1106
ParamSetItem::values 1106

```
<ParamSet Methods> ≡
    void ParamSet::AddFloat(const std::string &name, const Float *values,
                           int nValues) {
        EraseFloat(name);
        floats.emplace_back(new ParamsetItem<Float>(name, values, nValues));
    }
```

We won't include the rest of the methods to add data to the ParamSet, but we do include their prototypes here for reference. The erasure methods are also straightforward and won't be included here.

<ParamSet Public Methods> ≡

1105

```
void AddInt(const std::string &, const int *, int nValues);
void AddBool(const std::string &, const bool *, int nValues);
void AddPoint2f(const std::string &, const Point2f *, int nValues);
void AddVector2f(const std::string &, const Vector2f *, int nValues);
void AddPoint3f(const std::string &, const Point3f *, int nValues);
void AddVector3f(const std::string &, const Vector3f *, int nValues);
void AddNormal3f(const std::string &, const Normal3f *, int nValues);
void AddString(const std::string &, const std::string *, int nValues);
void AddTexture(const std::string &, const std::string &);
```

A number of different methods for adding spectral data are provided, making it easy for this data to be supplied with a variety of representations. The RGB and XYZ variants take 3 floating-point values for each spectrum. AddBlackbodySpectrum() takes pairs of temperature in Kelvins and a scale factor; it uses BlackbodyNormalized() to compute the SPD, which it scales with the given scale. Finally, AddSampledSpectrumFiles() reads SPDs from files on disk; both it and AddSampledSpectrum() construct a piecewise linear SPD given pairs of wavelengths and SPD values at each wavelength.

<ParamSet Public Methods> +≡

1105

```
void AddRGBSpectrum(const std::string &, const Float *, int nValues);
void AddXYZSpectrum(const std::string &, const Float *, int nValues);
void AddBlackbodySpectrum(const std::string &, const Float *,
                         int nValues);
void AddSampledSpectrumFiles(const std::string &, const char **,
                            int nValues);
void AddSampledSpectrum(const std::string &, const Float *, int nValues);
```

BlackbodyNormalized() 711

Float 1062

Normal3f 71

ParamSet 1105

ParamsetItem 1106

Point2f 68

Point3f 68

Vector2f 60

Vector3f 60

B.1.3 LOOKING UP VALUES IN THE PARAMETER SET

To retrieve a parameter value from a set, it is necessary to loop through the entries of the requested type and return the appropriate value, if any. There are two versions of the lookup method for each parameter type: a simple one for parameters that have a single data value, and a more general one that returns a pointer to the possibly multiple values of array parameter types. The first method mostly serves to reduce the amount of code needed in routines that retrieve parameter values.

The methods that look up a single item (e.g., `FindOneFloat()`) take the name of the parameter and a default value. If the parameter is not found, the default value is returned. This makes it easy to write initialization code like

```
Float radius = params.FindOneFloat("radius", 1.f);
```

In this case, it is not an error if the user didn't provide a "radius" parameter value; the default value will be used instead. If calling code wants to detect a missing parameter and issue an error, the appropriate second variant of lookup method should be used, since those methods return a `nullptr` value if the parameter isn't found.

```
(ParamSet Methods) +≡
Float ParamSet::FindOneFloat(const std::string &name, Float d) const {
    for (const auto &f : floats)
        if (f->name == name && f->nValues == 1) {
            f->lookedUp = true;
            return f->values[0];
        }
    return d;
}
```

We won't include the declarations of the analogous methods for the remaining types here (`FindOneInt()`, `FindOnePoint3f()`, and so forth); they all follow the same form as `FindOneFloat()`—each takes a parameter name and a default value and returns a value of the corresponding type.

The second kind of lookup method returns a pointer to the data if the data is present and returns the number of values in `n`.

```
(ParamSet Methods) +≡
const Float *ParamSet::FindFloat(const std::string &name, int *n) const {
    for (const auto &f : floats)
        if (f->name == name) {
            *n = f->nValues;
            f->lookedUp = true;
            return f->values.get();
        }
    return nullptr;
}
```

The general lookup functions for the other types follow the same form and so won't be included here.

Because the user may misspell parameter names in the scene description file, the `ParamSet` also provides a `ReportUnused()` function, not included here, that goes through the parameter set and reports if any of the parameters present were never looked up, checking the `ParamsetItem::lookedUp` member variable. For any items where this variable is `false`, it is likely that the user has given an incorrect parameter.

```
(ParamSet Public Methods) +≡
void ReportUnused() const;
```

The `ParamSet::Clear()` method clears all of the individual parameter vectors. The corresponding `ParamSetItems` will in turn be freed if their reference count goes to 0.

(ParamSet Public Methods) +≡
 void Clear();

1105

B.2 INITIALIZATION AND RENDERING OPTIONS

We now have the machinery to describe the routines that make up the rendering API. Before any other API functions can be called, the rendering system must be initialized by a call to `pbrtInit()`. Similarly, when rendering is done, `pbrtCleanup()` should be called; this handles final cleanup of the system. The definitions of these two functions will be filled in at a number of points throughout the rest of this appendix.

A few system-wide options are passed to `pbrtInit()` using the `Options` structure here.

(Global Forward Declarations) +≡
 struct Options {
 int nThreads = 0;
 bool quickRender = false;
 bool quiet = false, verbose = false;
 std::string imageFile;
};

(API Function Definitions) ≡
void pbrtInit(const Options &opt) {
 PbrtOptions = opt;
(API Initialization 1110)
(General pbrt Initialization 324)
}

The options are stored in a global variable for easy access by other parts of the system. This variable is only used in a read-only fashion by the system after initialization in `pbrtInit()`.

(API Global Variables) ≡
 Options PbrtOptions;

(API Function Definitions) +≡
void pbrtCleanup() {
(API Cleanup 1111)
}

After the system has been initialized, a subset of the API routines is available. Legal calls at this point are those that set general rendering options like the camera and sampler properties, the type of film to be used, and so on, but the user is not yet allowed to start to describe the lights, shapes, and materials in the scene.

Options 1109
`ParamSet::Clear()` 1109
`PbrtOptions` 1109
`pbrtWorldBegin()` 1117

After the overall rendering options have been set, the `pbrtWorldBegin()` function locks them in; it is no longer legal to call the routines that set them. At this point, the user can begin to describe the geometric primitives and lights that are in the scene. This separation

of global versus scene-specific information can help simplify the implementation of the renderer. For example, consider a spline patch shape that tessellates itself into triangles. This shape might compute the required size of its generated triangles based on the area of the screen that it covers. If the camera's position and image resolution are guaranteed not to change after the shape is created, then the shape can potentially do the tessellation work immediately at creation time.

Once the scene has been fully specified, the `pbrtWorldEnd()` routine is called. At this point, the renderer knows that the scene description is complete and that rendering can begin. The image will be rendered and written to a file before `pbrtWorldEnd()` returns. The user may then specify new options for another frame of an animation, and then another `pbrtWorldBegin()/pbrtWorldEnd()` block to describe the geometry for the next frame, repeating as many times as desired. The remainder of this section will discuss the routines related to setting rendering options. Section B.3 describes the routines for specifying the scene inside the world block.

B.2.1 STATE TRACKING

There are three distinct states that the renderer's API can be in:

- *Uninitialized*: Before `pbrtInit()` has been called or after `pbrtCleanup()` has been called, no other API calls are legal.
- *Options block*: Outside a `pbrtWorldBegin()` and `pbrtWorldEnd()` pair, scene-wide global options may be set.
- *World block*: Inside a `pbrtWorldBegin()` and `pbrtWorldEnd()` pair, the scene may be described.

The module static variable `currentApiState` starts out with the value `APIState::Uninitialized`, indicating that the API system hasn't yet been initialized. Its value is updated appropriately by `pbrtInit()`, `pbrtWorldBegin()`, and `pbrtCleanup()`.

(API Static Data) ≡

```
enum class APIState { Uninitialized, OptionsBlock, WorldBlock };
static APIState currentApiState = APIState::Uninitialized;
```

Now we can start to define the implementation of `pbrtInit()`. `pbrtInit()` first makes sure that it hasn't already been called and then sets the `currentApiState` variable to `OptionsBlock` to indicate that the scene-wide options can be specified.

(API Initialization) ≡

```
if (currentApiState != APIState::Uninitialized)
    Error("pbrtInit() has already been called.");
currentApiState = APIState::OptionsBlock;
```

1109

currentApiState 1110
`Error()` 1068
`pbrtCleanup()` 1109
`pbrtInit()` 1109
`pbrtWorldBegin()` 1117
`pbrtWorldEnd()` 1129

Similarly, `pbrtCleanup()` makes sure that `pbrtInit()` has been called and that we're not in the middle of a `pbrtWorldBegin()/pbrtWorldEnd()` block before resetting the state to the uninitialized state.

```
<API Cleanup> ≡ 1109
    if (currentApiState == APIState::Uninitialized)
        Error("pbrtCleanup() called without pbrtInit().");
    else if (currentApiState == APIState::WorldBlock)
        Error("pbrtCleanup() called while inside world block.");
    currentApiState = APIState::Uninitialized;
```

All API functions that are only valid in particular states invoke a state verification macro like `VERIFY_INITIALIZED()`, to make sure that `currentApiState` holds an appropriate value. If the states don't match, an error message is printed and the function immediately returns. (Note that this check must be implemented as a macro rather than a separate function so that its return statement causes the calling function itself to return.)

```
<API Macros> ≡
#define VERIFY_INITIALIZED(func) \
    if (currentApiState == APIState::Uninitialized) { \
        Error("pbrtInit() must be before calling \"%s()\\". " \
              "Ignoring.", func); \
        return; \
    } else /* swallow trailing semicolon */
```

The implementations of `VERIFY_OPTIONS()` and `VERIFY_WORLD()` are analogous.

B.2.2 TRANSFORMATIONS

As the scene is being described, `pbrt` maintains *current transformation matrices* (CTMs), one for each of a number of points in time. If the transformations are different, then they describe an animated transformation. (Recall, for example, that the `AnimatedTransform` class defined in Section 2.9.3 stores two transformation matrices for two given times.) A number of API calls are available to modify the CTMs; when objects like shapes, cameras, and lights are created, the CTMs are passed to their constructor to define the transformation from their local coordinate system to world space.

The code below stores two CTMs in the module-local `curTransform` variable. They are represented by the `TransformSet` class, to be defined shortly, which stores a fixed number of transformations. The `activeTransformBits` variable is a bit-vector indicating which of the CTMs are active; the active transforms are updated when the transformation-related API calls are made, while the others are unchanged. This mechanism allows the user to modify some of the CTMs selectively in order to define animated transformations.

```
activeTransformBits 1111
AllTransformsBits 1112
AnimatedTransform 103
currentApiState 1110
curTransform 1111
Error() 1068
pbrtTransformTimes() 1115
TransformSet 1112
```

The implementation here just stores two transformation matrices, one that defines the CTM for the starting time (provided via the `pbrtTransformTimes()` call, defined in a few pages), and the other for the ending time.

```
(API Local Classes) ==
constexpr int MaxTransforms = 2;
constexpr int StartTransformBits = 1 << 0;
constexpr int EndTransformBits = 1 << 1;
constexpr int AllTransformsBits = (1 << MaxTransforms) - 1;
```

`TransformSet` is a small utility class that stores an array of transformations and provides some utility routines for managing them.

```
(API Local Classes) +≡
struct TransformSet {
    (TransformSet Public Methods 1112)
private:
    Transform t[MaxTransforms];
};
```

An accessor function is provided to access the individual `Transforms`.

```
(TransformSet Public Methods) ==
Transform &operator[](int i) {
    return t[i];
}
```

1112

The `Inverse()` method returns a new `TransformSet` that holds the inverses of the individual `Transforms`.

```
(TransformSet Public Methods) +≡
friend TransformSet Inverse(const TransformSet &ts) {
    TransformSet tInv;
    for (int i = 0; i < MaxTransforms; ++i)
        tInv.t[i] = Inverse(ts.t[i]);
    return tInv;
}
```

1112

The actual transformation functions are straightforward. Because the CTM is used both for the rendering options and the scene description phases, these routines only need to verify that `pbrtInit()` has been called.

```
(API Function Definitions) +≡
void pbrtIdentity() {
    VERIFY_INITIALIZED("Identity");
    FOR_ACTIVE_TRANSFORMS(curTransform[i] = Transform());
}
```

The `FOR_ACTIVE_TRANSFORMS()` macro encapsulates the logic for determining which of the CTMs is active and applying the given operation to those that are. The given statement is executed only for the active transforms.

```
(API Macros) +≡
#define FOR_ACTIVE_TRANSFORMS(expr) \
    for (int i = 0; i < MaxTransforms; ++i) \
        if (activeTransformBits & (1 << i)) { expr }
```

activeTransformBits 1111
 curTransform 1111
 FOR_ACTIVE_TRANSFORMS() 1112
 Inverse() 1081
 MaxTransforms 1112
 pbrtInit() 1109
 Transform 83
 TransformSet 1112
 VERIFY_INITIALIZED 1111

```
(API Function Definitions) +≡
    void pbrtTranslate(Float dx, Float dy, Float dz) {
        VERIFY_INITIALIZED("Translate");
        FOR_ACTIVE_TRANSFORMS(curTransform[i] =
            curTransform[i] * Translate(Vector3f(dx, dy, dz));)
    }
```

Most of the rest of the functions are similarly defined, so we will not show their definitions here. `pbrt` also provides `pbrtConcatTransform()` and `pbrtTransform()` functions to allow the user to specify an arbitrary matrix to postmultiply or replace the active CTM(s), respectively.

```
(API Function Declarations) ≡
    void pbrtRotate(Float angle, Float ax, Float ay, Float az);
    void pbrtScale(Float sx, Float sy, Float sz);
    void pbrtLookAt(Float ex, Float ey, Float ez,
                    Float lx, Float ly, Float lz,
                    Float ux, Float uy, Float uz);
    void pbrtConcatTransform(Float transform[16]);
    void pbrtTransform(Float transform[16]);
```

It can be useful to make a named copy of the CTM so that it can be referred to later. For example, to place a light at the camera's position, it is useful to first apply the transformation into the camera coordinate system, since then the light can just be placed at the origin (0, 0, 0). This way, if the camera position is changed and the scene is rerendered, the light will move with it. The `pbrtCoordinateSystem()` function copies the current `TransformSet` into the `namedCoordinateSystems` associative array, and `pbrtCoordSysTransform()` loads a named set of CTMs.

```
(API Static Data) +≡
    static std::map<std::string, TransformSet> namedCoordinateSystems;
```

```
(API Function Definitions) +≡
    void pbrtCoordinateSystem(const std::string &name) {
        VERIFY_INITIALIZED("CoordinateSystem");
        namedCoordinateSystems[name] = curTransform;
    }
```

curTransform 1111
 float 1062
 FOR_ACTIVE_TRANSFORMS() 1112
 namedCoordinateSystems 1113
 pbrtConcatTransform() 1113
 pbrtCoordinateSystem() 1113
 pbrtCoordSysTransform() 1113
 pbrtTransform() 1113
 TransformSet 1112
 Vector3f 60
 VERIFY_INITIALIZED 1111
 Warning() 1068

```
(API Function Definitions) +≡
    void pbrtCoordSysTransform(const std::string &name) {
        VERIFY_INITIALIZED("CoordSysTransform");
        if (namedCoordinateSystems.find(name) !=
            namedCoordinateSystems.end())
            curTransform = namedCoordinateSystems[name];
        else
            Warning("Couldn't find named coordinate system \"%s\"", name.c_str());
    }
```

Not all of the types in `pbrt` that take transformations support animated transformations. (Textures are one example (Section B.3.2); it's not worth the additional code complexity to support them, especially since the utility an animated texture transform brings isn't obvious.) For such cases, `WARN_IF_ANIMATED_TRANSFORM()` macro warns if the CTMs are different, indicating that an animated transformation has been specified.

```
(API Macros) +≡
#define WARN_IF_ANIMATED_TRANSFORM(func) \
do { if (curTransform.IsAnimated()) \
    Warning("Animated transformations set; ignoring for \"%s\" " \
           "and using the start transform only", func); \
} while (false) /* swallow trailing semicolon */
```

```
(TransformSet Public Methods) +≡
bool IsAnimated() const { 1112
    for (int i = 0; i < MaxTransforms - 1; ++i)
        if (t[i] != t[i + 1]) return true;
    return false;
}
```

B.2.3 OPTIONS

All of the rendering options that are set before the `pbrtWorldBegin()` call are stored in a `RenderOptions` structure. This structure contains public data members that are set by API calls and methods that help create objects used by the rest of `pbrt` for rendering.

```
(API Local Classes) +≡
struct RenderOptions {
    (RenderOptions Public Methods 1130)
    (RenderOptions Public Data 1115)
};
```

A single static instance of a `RenderOptions` structure is available to the rest of the API functions:

```
(API Static Data) +≡
static std::unique_ptr<RenderOptions> renderOptions;
```

When `pbrtInit()` is called, it allocates a `RenderOptions` structure that initially holds default values for all of its options:

```
(API Initialization) +≡
renderOptions.reset(new RenderOptions); 1109
```

The `renderOptions` variable is freed by `pbrtCleanup()`:

```
(API Cleanup) +≡
renderOptions.reset(nullptr); 1109
```

A few calls are available to set which of the CTMs should be active.

`curTransform` 1111
`MaxTransforms` 1112
`pbrtInit()` 1109
`pbrtWorldBegin()` 1117
`RenderOptions` 1114
`renderOptions` 1114
`TransformSet::IsAnimated()`
 1114
`TransformSet::t` 1112
`WARN_IF_ANIMATED_TRANSFORM()`
 1114

```
{API Function Definitions} +≡
void pbrtActiveTransformAll() {
    activeTransformBits = AllTransformsBits;
}
void pbrtActiveTransformEndTime() {
    activeTransformBits = EndTransformBits;
}
void pbrtActiveTransformStartTime() {
    activeTransformBits = StartTransformBits;
}
```

The two times at which the two CTMs are defined can be provided by calling the function `pbrtTransformTimes()`. By default, the start time is 0 and the end time is 1.

```
{API Function Definitions} +≡
void pbrtTransformTimes(Float start, Float end) {
    VERIFY_OPTIONS("TransformTimes");
    renderOptions->transformStartTime = start;
    renderOptions->transformEndTime = end;
}
```

{RenderOptions Public Data} ≡

```
Float transformStartTime = 0, transformEndTime = 1;
```

1114

The API functions for setting the rest of the rendering options are mostly similar in both their interface and their implementation. For example, `pbrtPixelFilter()` specifies the kind of Filter to be used for filtering image samples. It takes two parameters: a string giving the name of the filter to use and a ParamSet giving the parameters to the filter.

Note that an instance of the Filter class isn't created immediately upon a call to `pbrtPixelFilter();` instead, that function just stores the name of the filter and its parameters in `renderOptions`. There are two reasons for this approach: first, there may be a subsequent call to `pbrtPixelFilter()` before the start of the world block, specifying a different filter; the (small) cost of creating the first Filter would be wasted in this case.

activeTransformBits 1111
 AllTransformsBits 1112
 Camera 356
 EndTransformBits 1112
 Film 484
 Filter 474
 Float 1062
 ParamSet 1105
 pbrtTransformTimes() 1115
 renderOptions 1114
 RenderOptions::FilterName
 1116
 RenderOptions::FilterParams
 1116
 StartTransformBits 1112

Second, and more importantly, there are various object creation ordering dependencies imposed by the parameters taken by various constructors. For example, the Film constructor expects a pointer to the Filter being used, and the Camera constructor expects a pointer to the Film. Thus, the camera can't be created before the film, and the film can't be created before the filter. We don't want to require the user to specify the scene options in an order dictated by these internal details, so instead always just store object names and parameter sets until the end of the options block. (The Filter here could actually be created immediately, since it doesn't depend on other objects, but we follow the same approach to it for consistency.)

{API Function Definitions} +≡

```
void pbrtPixelFilter(const std::string &name, const ParamSet &params) {
    VERIFY_OPTIONS("PixelFilter");
    renderOptions->FilterName = name;
    renderOptions->FilterParams = params;
}
```

The default filter is set to the box filter. If no specific filter is specified in the scene description file, then because the default ParamSet has no parameter values, the filter will be created based on its default parameter settings.

(RenderOptions Public Data) +≡

```
std::string FilterName = "box";
ParamSet FilterParams;
```

1114

Most of the rest of the rendering-option-setting API calls are similar; they simply store their arguments in `renderOptions`. Therefore, we will only include the declarations of these functions here. The options controlled by each function should be apparent from its name; more information about the legal parameters to each of these routines can be found in the documentation of pbrt's input file format.

(API Function Declarations) +≡

```
void pbrtFilm(const std::string &type, const ParamSet &params);
void pbrtSampler(const std::string &name, const ParamSet &params);
void pbrtAccelerator(const std::string &name, const ParamSet &params);
void pbrtIntegrator(const std::string &name, const ParamSet &params);
```

`pbrtCamera()` is slightly different from the other options, since the camera-to-world transformation needs to be recorded. The CTM is used by `pbrtCamera()` to initialize this value, and the camera coordinate system transformation is also stored for possible future use by `pbrtCoordSysTransform()`.

(API Function Definitions) +≡

```
void pbrtCamera(const std::string &name, const ParamSet &params) {
    VERIFY_OPTIONS("Camera");
    renderOptions->CameraName = name;
    renderOptions->CameraParams = params;
    renderOptions->CameraToWorld = Inverse(curTransform);
    namedCoordinateSystems["camera"] = renderOptions->CameraToWorld;
}
```

The default camera uses a perspective projection.

(RenderOptions Public Data) +≡

```
std::string CameraName = "perspective";
ParamSet CameraParams;
TransformSet CameraToWorld;
```

1114

curTransform [1111](#)
HomogeneousMedium [689](#)
Inverse() [1081](#)
namedCoordinateSystems [1113](#)
ParamSet [1105](#)
pbrtCamera() [1116](#)
pbrtCoordSysTransform() [1113](#)
renderOptions [1114](#)
RenderOptions::CameraName
[1116](#)
RenderOptions::CameraParams
[1116](#)
RenderOptions::CameraToWorld
[1116](#)
TransformSet [1112](#)

B.2.4 MEDIA DESCRIPTION

Definitions of participating media in the scene are specified by `pbrtMakeNamedMedium()`. This function allows the user to associate a specific type of participation media (from Section 11.3) with an arbitrary name. For example,

```
MakeNamedMedium "highAlbedo" "string type" "homogeneous"
"color sigma_s" [5.0 5.0 5.0] "color sigma_a" [0.1 0.1 0.1]
```

creates a `HomogeneousMedium` instance with the name `highAlbedo`.

(API Function Declarations) +≡

```
void pbrtMakeNamedMedium(const std::string &name, const ParamSet &params);
```

The corresponding Medium instances are stored in an associative array for access later.

(RenderOptions Public Data) +≡

1114

```
std::map<std::string, std::shared_ptr<Medium>> namedMedia;
```

Once named media have been created, pbrtMediumInterface() allows specifying the current “inside” and “outside” media. For shapes, these specify the media inside and outside the shape’s surface, where the side of the shape where the surface normal is oriented outward is “outside.” For the camera and for light sources that don’t have geometry associated with them, the “inside” medium is ignored and “outside” gives the medium containing the object. The current medium is stored in the GraphicsState class, which will be introduced shortly.

Like the transformation-related API functions, both of these functions can be called from both the options and the world blocks; the former so that media can be specified for the camera and the latter so that media can be specified for the lights and shapes in the scene.

(API Function Definitions) +≡

```
void pbrtMediumInterface(const std::string &insideName,
                        const std::string &outsideName) {
    VERIFY_INITIALIZED("MediumInterface");
    graphicsState.currentInsideMedium = insideName;
    graphicsState.currentOutsideMedium = outsideName;
}
```

activeTransformBits 1111
 AllTransformsBits 1112
 APIState::WorldBlock 1110
 currentApiState 1110
 curTransform 1111
 GraphicsState 1118
 graphicsState 1119
 GraphicsState::
 currentInsideMedium
 1117
 GraphicsState::
 currentOutsideMedium
 1117
 MaxTransforms 1112
 Medium 684
 namedCoordinateSystems 1113
 ParamSet 1105
 pbrtMediumInterface() 1117
 pbrtWorldBegin() 1117
 Transform 83
 VERIFY_INITIALIZED 1111

(Graphics State) ≡

1118

```
std::string currentInsideMedium, currentOutsideMedium;
```

B.3 SCENE DEFINITION

After the user has set up the overall rendering options, the pbrtWorldBegin() call marks the start of the description of the shapes, materials, and lights in the scene. It sets the current rendering state to APIState::WorldBlock, resets the CTMs to identity matrices, and enables all of the CTMs.

(API Function Definitions) +≡

```
void pbrtWorldBegin() {
    VERIFY_OPTIONS("WorldBegin");
    currentApiState = APIState::WorldBlock;
    for (int i = 0; i < MaxTransforms; ++i)
        curTransform[i] = Transform();
    activeTransformBits = AllTransformsBits;
    namedCoordinateSystems["world"] = curTransform;
}
```

B.3.1 HIERARCHICAL GRAPHICS STATE

As the scene's lights, geometry, and participating media are specified, a variety of attributes can be set as well. In addition to the CTMs, these include information about textures and the current material. When a geometric primitive or light source is then added to the scene, the current attributes are used when creating the corresponding object. These data are all known as the *graphics state*.

It is useful for a rendering API to provide some functionality for managing the graphics state. pbrt has API calls that allow the current graphics state to be managed with an *attribute stack*; the user can push the current set of attributes, make changes to their values, and then later pop back to the previously pushed attribute values. For example, a scene description file might contain the following:

```
Material "matte"
AttributeBegin
    Material "plastic"
    Translate 5 0 0
    Shape "sphere" "float radius" [1]
AttributeEnd
Shape "sphere" "float radius" [1]
```

The first sphere is affected by the translation and is bound to the plastic material, while the second sphere is matte and isn't translated. Changes to attributes made inside a `pbrtAttributeBegin()`/`pbrtAttributeEnd()` block are forgotten at the end of the block. Being able to save and restore attributes in this manner is a classic idiom for scene description in computer graphics.

The graphics state is stored in the `GraphicsState` structure. As was done previously with `RenderOptions`, we'll be adding members to it throughout this section.

```
(API Local Classes) +≡
struct GraphicsState {
    (Graphics State Methods 1125)
    (Graphics State 1117)
};
```

When `pbrtInit()` is called, the current graphics state is initialized to hold default values.

```
(API Initialization) +≡
graphicsState = GraphicsState();
```

1109

<code>GraphicsState</code> 1118 <code>graphicsState</code> 1119 <code>pbrtAttributeBegin()</code> 1119 <code>pbrtAttributeEnd()</code> 1119 <code>pbrtInit()</code> 1109 <code>RenderOptions</code> 1114

A vector of `GraphicsStates` is used as a stack to perform hierarchical state management. When `pbrtAttributeBegin()` is called, the current `GraphicsState` is copied and pushed onto this stack. `pbrtAttributeEnd()` then simply pops the state from this stack.

```

⟨API Function Definitions⟩ +≡
    void pbrtAttributeBegin() {
        VERIFY_WORLD("AttributeBegin");
        pushedGraphicsStates.push_back(graphicsState);
        pushedTransforms.push_back(curTransform);
        pushedActiveTransformBits.push_back(activeTransformBits);
    }

⟨API Static Data⟩ +≡
    static GraphicsState graphicsState;
    static std::vector<GraphicsState> pushedGraphicsStates;
    static std::vector<TransformSet> pushedTransforms;
    static std::vector<uint32_t> pushedActiveTransformBits;

pbrtAttributeEnd() also verifies that we do not have attribute stack underflow by checking to see if the stack is empty.

⟨API Function Definitions⟩ +≡
    void pbrtAttributeEnd() {
        VERIFY_WORLD("AttributeEnd");
        if (!pushedGraphicsStates.size()) {
            Error("Unmatched pbrtAttributeEnd() encountered. "
                  "Ignoring it.");
            return;
        }
        graphicsState = pushedGraphicsStates.back();
        pushedGraphicsStates.pop_back();
        curTransform = pushedTransforms.back();
        pushedTransforms.pop_back();
        activeTransformBits = pushedActiveTransformBits.back();
        pushedActiveTransformBits.pop_back();
    }
}

```

activeTransformBits 1111
 curTransform 1111
 Error() 1068
 GraphicsState 1118
 graphicsState 1119
 pbrtAttributeBegin() 1119
 pbrtAttributeEnd() 1119
 pbrtTransformBegin() 1119
 pbrtTransformEnd() 1119
 pushedActiveTransformBits
 1119
 pushedGraphicsStates 1119
 pushedTransforms 1119
 TransformSet 1112

The API also provides `pbrtTransformBegin()` and `pbrtTransformEnd()` calls. These functions are similar to `pbrtAttributeBegin()` and `pbrtAttributeEnd()`, except that they only push and pop the CTMs. We frequently want to apply a transformation to a texture, but since the list of named textures is stored in the graphics state, we cannot use `pbrtAttributeBegin()` to save the transformation matrix. Since the implementations of `pbrtTransformBegin()` and `pbrtTransformEnd()` are very similar to `pbrtAttributeBegin()` and `pbrtAttributeEnd()`, respectively, they are not shown here.

```

⟨API Function Declarations⟩ +≡
    void pbrtTransformBegin();
    void pbrtTransformEnd();

```

B.3.2 TEXTURE AND MATERIAL PARAMETERS

Recall that all of the materials in `pbrt` use textures to describe all of their parameters. For example, the diffuse color of the matte material class is always obtained from a

texture, even if the material is intended to have a constant reflectivity (in which case a `ConstantTexture` is used).

Before a material can be created, it is necessary to create these textures to pass to the material creation procedures. Textures can be either explicitly created and later referred to by name or implicitly created on the fly to represent a constant parameter. These two methods of texture creation are hidden by the `TextureParams` class.

```
(TextureParams Declarations) ≡
class TextureParams {
public:
    (TextureParams Public Methods 1121)
private:
    (TextureParams Private Data 1120)
};
```

The `TextureParams` class stores references to associative arrays of previously defined named `Float` and `Spectrum` textures, as well as two references to `ParamSets` that will be searched for named textures. Its constructor, which won't be included here, just initializes these references from parameters passed to it.

```
(TextureParams Private Data) ≡ 1120
std::map<std::string, std::shared_ptr<Texture<Float>>> &floatTextures;
std::map<std::string,
         std::shared_ptr<Texture<Spectrum>>> &spectrumTextures;
const ParamSet &geomParams, &materialParams;
```

Here we will show the code for finding a texture of `Spectrum` type; the code for finding a `Float` texture is analogous. The `TextureParams::GetSpectrumTexture()` method takes a parameter name (e.g., "Kd"), as well as a default `Spectrum` value. If no texture has been explicitly specified for the parameter, a constant texture will be created that returns the default spectrum value.

Finding the texture is performed in several stages; the order of these stages is significant. First, the parameter list from the `Shape` for which a `Material` is being created is searched for a named reference to an explicitly defined texture. If no such texture is found, then the material parameters are searched. Finally, if no explicit texture has been found, the two parameter lists are searched in turn for supplied constant values. If no such constants are found, the default is used.

The order of these steps is crucial, because `pbrt` allows a shape to override individual elements of the material that is bound to it. For example, the user should be able to create a scene description that contains the lines

```
Material "matte" "color Kd" [ 1 0 0 ]
Shape "sphere" "color Kd" [ 0 1 0 ]
```

These two commands create a green matte sphere: because the shape's parameter list is searched first, the `Kd` parameter from the `Shape` will be used when the `MatteMaterial` constructor is called.

`ConstantTexture` 615
`Float` 1062
`Material` 577
`MatteMaterial` 578
`ParamSet` 1105
`Shape` 123
`Spectrum` 315
`Texture` 614
`TextureParams` 1120
`TextureParams::GetSpectrumTexture()` 1121

```

<TextureParams Method Definitions> ≡
    std::shared_ptr<Texture<Spectrum>>
    TextureParams::GetSpectrumTexture(const std::string &n,
                                      const Spectrum &def) const {
        std::string name = geomParams.FindTexture(n);
        if (name == "") name = materialParams.FindTexture(n);
        if (name != "") {
            if (spectrumTextures.find(name) != spectrumTextures.end())
                return spectrumTextures[name];
            else
                Error("Couldn't find spectrum texture named \"%s\" "
                      "for parameter \"%s\"", name.c_str(), n.c_str());
        }
        Spectrum val = materialParams.FindOneSpectrum(n, def);
        val = geomParams.FindOneSpectrum(n, val);
        return std::make_shared<ConstantTexture<Spectrum>>(val);
    }
}

```

Because an instance of the `TextureParams` class is passed to material creation routines that might need to access non-texture parameter values, we also provide ways to access the other parameter list types. These methods return parameters from the geometric parameter list, if found. Otherwise, the material parameter list is searched, and finally the default value is returned.

ConstantTexture 615
 Error() 1068
 Float 1062
 graphicsState 1119
 GraphicsState::floatTextures
 1122
 GraphicsState::
 spectrumTextures
 1122
 ParamSet 1105
 ParamSet::FindOneFloat()
 1108
 ParamSet::FindOneSpectrum()
 1108
 ParamSet::FindTexture() 1108
 pbrtTexture() 1121
 Spectrum 315
 Texture 614
 TextureParams 1120
 TextureParams::FindFloat()
 1121
 TextureParams::geomParams
 1120
 TextureParams::materialParams
 1120
 TextureParams::
 spectrumTextures
 1120

The `TextureParams::FindFloat()` method is shown here. The other access methods are similar and omitted.

```

<TextureParams Public Methods> ≡
    float FindFloat(const std::string &n, float d) const {
        return geomParams.FindOneFloat(n, materialParams.FindOneFloat(n, d));
    }

```

1120

B.3.3 SURFACE AND MATERIAL DESCRIPTION

The `pbrtTexture()` method creates a named texture that can be referred to later. In addition to the texture name, its *type* is specified. `pbrt` supports only “float” and “spectrum” as texture types. The supplied parameter list is used to create a `TextureParams` object, which will be passed to the desired texture’s creation routine.

<API Function Definitions> +≡
 void pbrtTexture(const std::string &name, const std::string &type,
 const std::string &texname, const ParamSet ¶ms) {
 VERIFY_WORLD("Texture");
 TextureParams tp(params, params, graphicsState.floatTextures,
 graphicsState.spectrumTextures);
 if (type == "float") {
 <Create Float texture and store in floatTextures 1122>
 }
 }

```

else if (type == "color" || type == "spectrum") {
    (Create color texture and store in spectrumTextures)
}
else
    Error("Texture type \"%s\" unknown.", type.c_str());
}

```

Creating the texture is simple. This function first checks to see if a texture of the same name and type already exists and issues a warning if so. Then, the `MakeFloatTexture()` routine calls the creation function for the appropriate Texture implementation, and the returned texture class is added to the `GraphicsState::floatTextures` associative array. The code for creating a spectrum texture is similar and not shown.

(Create Float texture and store in floatTextures) ≡ 1121

```

if (graphicsState.floatTextures.find(name) !=
    graphicsState.floatTextures.end())
    Info("Texture \"%s\" being redefined", name.c_str());
WARN_IF_ANIMATED_TRANSFORM("Texture");
std::shared_ptr<Texture<Float>> ft =
    MakeFloatTexture(texname, curTransform[0], tp);
if (ft) graphicsState.floatTextures[name] = ft;

```

(Graphics State) +≡ 1118

```

std::map<std::string, std::shared_ptr<Texture<Float>>> floatTextures;
std::map<std::string,
         std::shared_ptr<Texture<Spectrum>>> spectrumTextures;

```

The current material is specified by a call to `pbrtMaterial()`. Its `ParamSet` is stored until a `Material` object needs to be created later when a shape is specified.

(API Function Declarations) +≡

```

void pbrtMaterial(const std::string &name, const ParamSet &params);

```

The default material is matte.

(Graphics State) +≡ 1118

```

ParamSet materialParams;
std::string material = "matte";

```

pbrt also supports the notion of creating a `Material` with a given set of parameters and then associating an arbitrary name with the combination of material and parameter settings. `pbrtMakeNamedMaterial()` creates such an association, and `pbrtNamedMaterial()` sets the current material and material parameters based on a previously defined named material.

(API Function Declarations) +≡

```

void pbrtMakeNamedMaterial(const std::string &name,
                           const ParamSet &params);
void pbrtNamedMaterial(const std::string &name);

```

curTransform 1111
 Float 1062
 graphicsState 1119
 GraphicsState::floatTextures
 1122
 Info() 1068
 Material 577
 ParamSet 1105
 pbrtMakeNamedMaterial() 1122
 pbrtMaterial() 1122
 pbrtNamedMaterial() 1122
 Spectrum 315
 Texture 614
 WARN_IF_ANIMATED_TRANSFORM()
 1114

```
<Graphics State> +≡
    std::map<std::string, std::shared_ptr<Material>> namedMaterials;
    std::string currentNamedMaterial;
```

1118

B.3.4 LIGHT SOURCES

pbrt's API provides two ways for the user to specify light sources for the scene. The first, `pbrtLightSource()`, defines a light source that doesn't have geometry associated with it (e.g., a point light or a directional light).

(API Function Definitions) +≡

```
void pbrtLightSource(const std::string &name, const ParamSet &params) {
    VERIFY_WORLD("LightSource");
    WARN_IF_ANIMATED_TRANSFORM("LightSource");
    MediumInterface mi = graphicsState.CreateMediumInterface();
    std::shared_ptr<Light> lt =
        MakeLight(name, params, curTransform[0], mi);
    if (!lt)
        Error("LightSource: light type \"%s\" unknown.", name.c_str());
    else
        renderOptions->lights.push_back(lt);
}
```

1114

(RenderOptions Public Data) +≡

```
std::vector<std::shared_ptr<Light>> lights;
```

The second API call to describe light sources, `pbrtAreaLightSource()`, specifies an area light source. All shape specifications that appear between an area light source call up to the end of the current attribute block are treated as emissive. Thus, when an area light is specified via `pbrtAreaLightSource()`, it can't be created immediately since the shapes to follow are needed to define the light source's geometry. Therefore, this function just saves the name of the area light source type and the parameters given to it.

curTransform 1111
 Error() 1068
 graphicsState 1119
 GraphicsState::
 CreateMediumInterface() 1125
 Light 714
 MakeLight() 1131
 Material 577
 MediumInterface 684
 ParamSet 1105
 pbrtAreaLightSource() 1123
 pbrtLightSource() 1123
 pbrtShape() 1124
 renderOptions 1114
 RenderOptions::lights 1123
 Shape 123

1118

(Graphics State) +≡

```
ParamSet areaLightParams;
std::string areaLight;
```

(API Function Definitions) +≡

```
void pbrtAreaLightSource(const std::string &name,
                        const ParamSet &params) {
    VERIFY_WORLD("AreaLightSource");
    graphicsState.areaLight = name;
    graphicsState.areaLightParams = params;
}
```

B.3.5 SHAPES

The `pbrtShape()` function creates one or more new `Shape` objects and adds them to the scene. This function is relatively complicated, in that it has to create area light sources if an area light is being defined, it has to handle animated and static shapes differently, and it also has to deal with creating object instances when needed.

```
(API Function Definitions) +≡
void pbtrShape(const std::string &name, const ParamSet &params) {
    VERIFY_WORLD("Shape");
    std::vector<std::shared_ptr<Primitive>> prims;
    std::vector<std::shared_ptr<AreaLight>> areaLights;
    if (!curTransform.IsAnimated()) {
        <Initialize prims and areaLights for static shape 1124>
    } else {
        <Initialize prims and areaLights for animated shape 1126>
    }
    <Add prims and areaLights to scene or current instance 1127>
}
```

Shapes that are animated are represented with `TransformedPrimitives`, which include extra functionality to use `AnimatedTransforms`, while shapes that aren't animated use `GeometricPrimitives`. Therefore, there are two code paths here for those two cases.

The static shape case is mostly a matter of creating the appropriate `Shapes`, `Material`, and `MediumInterface` to make corresponding `GeometricPrimitives`.

```
<Initialize prims and areaLights for static shape> ≡ 1124
(Create shapes for shape name 1124)
std::shared_ptr<Material> mtl = graphicsState.CreateMaterial(params);
params.ReportUnused();
MediumInterface mi = graphicsState.CreateMediumInterface();
for (auto s : shapes) {
    <Possibly create area light for shape 1125>
    prims.push_back(
        std::make_shared<GeometricPrimitive>(s, mtl, area, mi));
}
```

AnimatedTransform 103
 AreaLight 734
 curTransform 1111
 GeometricPrimitive 250
 graphicsState 1119
 GraphicsState::
 CreateMaterial() 1125
 GraphicsState::
 CreateMediumInterface() 1125
 MakeShapes() 1125
 Material 577
 MediumInterface 684
 ParamSet 1105
 ParamSet::ReportUnused() 1108
 Primitive 248
 Shape 123
 Transform 83
 TransformCache 1124
 transformCache 1125
 TransformCache::Lookup() 1124
 TransformedPrimitive 252
 TransformSet::IsAnimated() 1114

The code below uses a `TransformCache` (defined shortly), which allocates and stores a single `Transform` pointer for each unique transformation that is passed to its `Lookup()` method. In this way, if many shapes in the scene have the same transformation matrix, a single `Transform` pointer can be shared among all of them. `MakeShapes()` then handles the details of creating the shape or shapes corresponding to the given shape name, passing the `ParamSet` along to the shape's creation routine.

```
(Create shapes for shape name)> ≡ 1124
Transform *ObjToWorld, *WorldToObj;
transformCache.Lookup(curTransform[0], &ObjToWorld, &WorldToObj);
std::vector<std::shared_ptr<Shape>> shapes =
    MakeShapes(name, ObjToWorld, WorldToObj,
               graphicsState.reverseOrientation, params);
if (shapes.size() == 0) return;
```

`TransformCache` is a small wrapper around an associative array from transformations to pairs of `Transform` pointers; the first pointer is equal to the transform, and the second is its inverse. The `Lookup()` method just looks for the given transformation in the cache,

allocates space for it and stores it and its inverse if not found, and returns the appropriate pointers.

```
(TransformCache Private Data) ≡
    std::map<Transform, std::pair<Transform *, Transform *>> cache;
    MemoryArena arena;

(API Static Data) +≡
    static TransformCache transformCache;
```

The `MakeShapes()` function takes the name of the shape to be created, the CTMs, and the `ParamSet` for the new shape. It calls an appropriate shape creation function based on the shape name provided (e.g., for “sphere,” it calls `CreateSphereShape()`, which is defined in Section B.4). The shape creation routines may return multiple shapes; for triangle meshes, for example, the creation routine returns a vector of `Triangles`. The implementation of this function is straightforward, so we won’t include it here.

```
(API Forward Declarations) ≡
    std::vector<std::shared_ptr<Shape>> MakeShapes(const std::string &name,
                                                    const Transform *ObjectToWorld, const Transform *WorldToObject,
                                                    bool reverseOrientation, const ParamSet &paramSet);
```

The `Material` for the shape is created by the `MakeMaterial()` call; its implementation is analogous to that of `MakeShapes()`. If the specified material cannot be found (usually due to a typo in the material name), a matte material is created and a warning is issued.

AreaLight 734
`CreateSphereShape()` 1131
`curTransform` 1111
`GeometricPrimitive` 250
`graphicsState` 1119
`GraphicsState::areaLight` 1123
`GraphicsState::areaLightParams` 1123
`MakeShapes()` 1125
`Material` 577
`MediumInterface` 684
`MemoryArena` 1074
`ParamSet` 1105
`pbrtAreaLightSource()` 1123
`pbrtMediumInterface()` 1117
`Shape` 123
`Transform` 83
`TransformCache` 1124
`TransformedPrimitive` 252
`Triangle` 156

(Graphics State Methods) ≡ 1118
`std::shared_ptr<Material> CreateMaterial(const ParamSet ¶ms);`

Following the same basic approach, `CreateMediumInterface()` creates a `MediumInterface` based on the current named “inside” and “outside” media established with `pbrtMediumInterface()`.

(Graphics State Methods) +≡ 1118
`MediumInterface CreateMediumInterface();`

If an area light has been set in the current graphics state by `pbrtAreaLightSource()`, the new shape is an emitter and an `AreaLight` needs to be made for it.

```
(Possibly create area light for shape) ≡ 1124
    std::shared_ptr<AreaLight> area;
    if (graphicsState.areaLight != "") {
        area = MakeAreaLight(graphicsState.areaLight, curTransform[0],
                             mi, graphicsState.areaLightParams, s);
        areaLights.push_back(area);
    }
```

If the transformation matrices are animated, the task is a little more complicated. After `Shape` and `GeometricPrimitive` creation, a `TransformedPrimitive` is created to hold the shape or shapes that were created.

<i>(Initialize prims and areaLights for animated shape) ≡</i> <i>(Create initial shape or shapes for animated shape 1126)</i> <i>(Create GeometricPrimitive(s) for animated shape 1126)</i> <i>(Create single TransformedPrimitive for prims 1126)</i>	1124
---	------

Because the Shape class doesn't handle animated transformations, the initial shape or shapes for animated primitives are created with identity transformations. All of the details related to the shape's transformation will be managed with the Transformed Primitive that ends up holding the shape. Animated transformations for light sources aren't currently supported in pbrt; thus, if an animated transform has been specified with an area light source, a warning is issued here.

<i>(Create initial shape or shapes for animated shape) ≡</i> <pre>if (graphicsState.areaLight != "") Warning("Ignoring currently set area light when creating " "animated shape"); Transform *identity; transformCache.Lookup(Transform(), &identity, nullptr); std::vector<std::shared_ptr<Shape>> shapes = MakeShapes(name, identity, identity, graphicsState.reverseOrientation, params); if (shapes.size() == 0) return;</pre>	1126
--	------

Given the initial set of shapes, it's straightforward to create a GeometricPrimitive for each of them.

<i>(Create GeometricPrimitive(s) for animated shape) ≡</i> <pre>std::shared_ptr<Material> mtl = graphicsState.CreateMaterial(params); params.ReportUnused(); MediumInterface mi = graphicsState.CreateMediumInterface(); for (auto s : shapes) prims.push_back(std::make_shared<GeometricPrimitive>(s, mtl, nullptr, mi));</pre>	1126
--	------

BVHAccel 256
GeometricPrimitive 250
graphicsState 1119
GraphicsState::areaLight
1123

GraphicsState::
CreateMaterial()
1125
MakeShapes() 1125
Material 577
MediumInterface 684
ParamSet::ReportUnused()
1108

Primitive 248
Shape 123
Transform 83
transformCache 1125
TransformCache::Lookup()
1124
TransformedPrimitive 252
Warning() 1068

<i>(Create single TransformedPrimitive for prims) ≡</i> <i>(Get animatedObjectToWorld transform for shape 1127)</i> <pre>if (prims.size() > 1) { std::shared_ptr<Primitive> bvh = std::make_shared<BVHAccel>(prims); prims.clear(); prims.push_back(bvh); } prims[0] = std::make_shared<TransformedPrimitive>(prims[0], animatedObjectToWorld);</pre>	1126
--	------

The `TransformCache` is used again, here to get transformations for the start and end time, which are then passed to the `AnimatedTransform` constructor.

```
(Get animatedObjectToWorld transform for shape) ≡ 1126
    Transform *ObjToWorld[2];
    transformCache.Lookup(curTransform[0], &ObjToWorld[0], nullptr);
    transformCache.Lookup(curTransform[1], &ObjToWorld[1], nullptr);
    AnimatedTransform animatedObjectToWorld(
        ObjToWorld[0], renderOptions->transformStartTime,
        ObjToWorld[1], renderOptions->transformEndTime);
```

If the user is in the middle of defining an object instance, `pbrtObjectBegin()` (defined in the following section) will have set the `currentInstance` member of `renderOptions` to point to a vector that is collecting the shapes that define the instance. In that case, the new shape or shapes are added to that array. Otherwise, the `RenderOptions::primitives` array is used—this array will eventually be passed to the `Scene` constructor. If it is also an area light, the corresponding `areaLights` are also added to the `RenderOptions::lights` array, just as `pbrtLightSource()` does.

```
(Add prims and areaLights to scene or current instance) ≡ 1124
if (renderOptions->currentInstance) {
    if (areaLights.size())
        Warning("Area lights not supported with object instancing");
    renderOptions->currentInstance->insert(
        renderOptions->currentInstance->end(), prims.begin(),
        prims.end());
} else {
    renderOptions->primitives.insert(renderOptions->primitives.end(),
        prims.begin(), prims.end());
    if (areaLights.size())
        renderOptions->lights.insert(renderOptions->lights.end(),
            areaLights.begin(), areaLights.end());
}
```

{RenderOptions Public Data} +≡ 1114
`std::vector<std::shared_ptr<Primitive>> primitives;`

B.3.6 OBJECT INSTANCING

All shapes that are specified between a `pbrtObjectBegin()` and `pbrtObjectEnd()` pair are used to create a named object instance (see the discussion of object instancing and the `TransformedPrimitive` class in Section 4.1.2). `pbrtObjectBegin()` sets `RenderOptions::currentInstance` so that subsequent `pbrtShape()` calls can add the shape to this instance's vector of primitive references. This function also pushes the graphics state, so that any changes made to the CTMs or other state while defining the instance don't last beyond the instance definition.

```

⟨API Function Definitions⟩ +≡
void pbrtObjectBegin(const std::string &name) {
    pbrtAttributeBegin();
    if (renderOptions->currentInstance)
        Error("ObjectBegin called inside of instance definition");
    renderOptions->instances[name] =
        std::vector<std::shared_ptr<Primitive>>();
    renderOptions->currentInstance = &renderOptions->instances[name];
}

⟨RenderOptions Public Data⟩ +≡
1114
std::map<std::string, std::vector<std::shared_ptr<Primitive>>> instances;
std::vector<std::shared_ptr<Primitive>> *currentInstance = nullptr;

⟨API Function Definitions⟩ +≡
void pbrtObjectEnd() {
    VERIFY_WORLD("ObjectEnd");
    if (!renderOptions->currentInstance)
        Error("ObjectEnd called outside of instance definition");
    renderOptions->currentInstance = nullptr;
    pbrtAttributeEnd();
}

```

When an instance is used in the scene, the instance’s vector of Primitives needs to be found in the RenderOptions::instances map, a TransformedPrimitive created, and the instance added to the scene. Note that the TransformedPrimitive constructor takes the current transformation matrix from the time when pbrtObjectInstance() is called. The instance’s complete world transformation is the composition of the CTM when it is instantiated with the CTM when it was originally created.

pbrtObjectInstance() first does some error checking to make sure that the instance is not being used inside the definition of another instance and also that the named instance has been defined. The error checking is simple and not shown here.

```

⟨API Function Definitions⟩ +≡
void pbrtObjectInstance(const std::string &name) {
    VERIFY_WORLD("ObjectInstance");
    ⟨Perform object instance error checking⟩
    std::vector<std::shared_ptr<Primitive>> &in =
        renderOptions->instances[name];
    if (in.size() == 0) return;
    if (in.size() > 1) {
        ⟨Create aggregate for instance Primitives 1129⟩
    }
    ⟨Create animatedInstanceToWorld transform for instance 1129⟩
    std::shared_ptr<Primitive> prim(
        std::make_shared<TransformedPrimitive>(in[0],
                                              animatedInstanceToWorld));
    renderOptions->primitives.push_back(prim);
}

```

Error() 1068
 pbrtAttributeBegin() 1119
 pbrtAttributeEnd() 1119
 pbrtObjectInstance() 1128
 Primitive 248
 renderOptions 1114
 RenderOptions::
 currentInstance
 1128
 RenderOptions::instances
 1128
 RenderOptions::primitives
 1127
 TransformedPrimitive 252

```
<Create animatedInstanceToWorld transform for instance> ≡ 1128
    Transform *InstanceToWorld[2];
    transformCache.Lookup(curTransform[0], &InstanceToWorld[0], nullptr);
    transformCache.Lookup(curTransform[1], &InstanceToWorld[1], nullptr);
    AnimatedTransform animatedInstanceToWorld(InstanceToWorld[0],
        renderOptions->transformStartTime,
        InstanceToWorld[1], renderOptions->transformEndTime);
```

If there is more than one primitive in an instance, then an aggregate needs to be built for it. This must be done here rather than in the `TransformedPrimitive` constructor so that the resulting aggregate will be reused if this instance is used multiple times in the scene.

```
<Create aggregate for instance Primitives> ≡ 1128
    std::shared_ptr<Primitive> accel(
        MakeAccelerator(renderOptions->AcceleratorName, in,
                        renderOptions->AcceleratorParams));
    if (!accel) accel = std::make_shared<BVHAccel>(in);
    in.erase(in.begin(), in.end());
    in.push_back(accel);
```

B.3.7 WORLD END AND RENDERING

When `pbrtWorldEnd()` is called, the scene has been fully specified and rendering can begin. This routine makes sure that there aren't excess graphics state structures pushed on the state stack (issuing a warning if so), creates the `Scene` and `Integrator` objects, and then calls the `Integrator::Render()` method.

```
AnimatedTransform 103
BVHAccel 256
curTransform 1111
Integrator 25
Integrator::Render() 25
Primitive 248
pushedGraphicsStates 1119
pushedTransforms 1119
renderOptions 1114
RenderOptions:::
    MakeIntegrator() 1130
    MakeScene() 1130
    Warning() 1068
    Warning("Missing end to pbrtAttributeBegin()"); 1115
    Warning("Missing end to pbrtTransformBegin()"); 1115
    transformCache 1125
    TransformedPrimitive 252
    Warning() 1068
```

```
<API Function Definitions> +≡
void pbrtWorldEnd() {
    VERIFY_WORLD("WorldEnd");
    <Ensure there are no pushed graphics states 1129>
    <Create scene and render 1130>
    <Clean up after rendering 1130>
}
```

If there are graphics states and/or transformations remaining on the respective stacks, a warning is issued for each one:

```
<Ensure there are no pushed graphics states> ≡ 1129
while (pushedGraphicsStates.size()) {
    Warning("Missing end to pbrtAttributeBegin()");
    pushedGraphicsStates.pop_back();
    pushedTransforms.pop_back();
}
while (pushedTransforms.size()) {
    Warning("Missing end to pbrtTransformBegin()");
    pushedTransforms.pop_back();
}
```

Now the `RenderOptions::MakeIntegrator()` and `RenderOptions::MakeScene()` methods can create the corresponding objects based on the settings provided by the user.

```
(Create scene and render) ≡ 1129
    std::unique_ptr<Integrator> integrator(renderOptions->MakeIntegrator());
    std::unique_ptr<Scene> scene(renderOptions->MakeScene());
    if (scene && integrator)
        integrator->Render(*scene);
    TerminateWorkerThreads();
```

Creating the Scene object is mostly a matter of creating the Aggregate for all of the primitives and calling the Scene constructor. The MakeAccelerator() function isn't included here; it's similar in structure to MakeShapes() as far as using the string passed to it to determine which accelerator construction function to call.

```
(API Function Definitions) +≡
Scene *RenderOptions::MakeScene() {
    std::shared_ptr<Primitive> accelerator =
        MakeAccelerator(AcceleratorName, primitives, AcceleratorParams);
    if (!accelerator)
        accelerator = std::make_shared<BVHAccel>(primitives);
    Scene *scene = new Scene(accelerator, lights);
    <Erase primitives and lights from RenderOptions 1130>
    return scene;
}
```

After the scene has been created, RenderOptions clears the vectors of primitives and lights. This ensures that if a subsequent scene is defined then the scene description from this frame isn't inadvertently included.

```
(Erase primitives and lights from RenderOptions) ≡ 1130
    primitives.erase(primitives.begin(), primitives.end());
    lights.erase(lights.begin(), lights.end());
```

Integrator creation in the MakeIntegrator() method is again similar to how shapes and other named objects are created; the string name is used to dispatch to an object-specific creation function.

```
(RenderOptions Public Methods) ≡ 1114
Integrator *MakeIntegrator() const;
```

Once rendering is complete, the API transitions back to the “options block” rendering state, prints out any statistics gathered during rendering, and clears the CTMs and named coordinate systems so that the next frame, if any, starts with a clean slate.

```
(Clean up after rendering) ≡ 1129
    currentApiState = APIState::OptionsBlock;
    ReportThreadStats();
    if (PbrtOptions.quiet == false) {
        PrintStats(stdout);
        ReportProfilerResults(stdout);
    }
```

activeTransformBits 1111
 Aggregate 255
 AllTransformsBits 1112
 BVHAccel 256
 currentApiState 1110
 curTransform 1111
 Integrator 25
 Integrator::Render() 25
 MakeShapes() 1125
 MaxTransforms 1112
 namedCoordinateSystems 1113
 PbrtOptions 1109
 Primitive 248
 RenderOptions 1114
 renderOptions 1114
 RenderOptions::lights 1123
 RenderOptions::
 MakeIntegrator()
 1130
 RenderOptions::MakeScene()
 1130
 RenderOptions::primitives
 1127
 ReportThreadStats() 1097
 Scene 23
 TerminateWorkerThreads()
 1093
 Transform 83

```

for (int i = 0; i < MaxTransforms; ++i)
    curTransform[i] = Transform();
activeTransformBits = AllTransformsBits;
namedCoordinateSystems.erase(namedCoordinateSystems.begin(),
    namedCoordinateSystems.end());

```

B.4 ADDING NEW OBJECT IMPLEMENTATIONS

We will briefly review the overall process that `pbrt` uses to create instances of implementations of the various abstract interface classes like `Shapes`, `Cameras`, `Integrators`, etc., at run time. We will focus on the details for the `Shape` class, since the other types are handled similarly.

When the API needs to create a shape, it has the string name of the shape and the `ParamSet` that represents the corresponding information in the input file. These need to be used together to create a specific instance of the named shape. `MakeShapes()` has a series of `if` tests to determine which shape creation function to call; the one for `Spheres`, `CreateSphereShape()`, is shown here:

```

<Sphere Method Definitions> +≡
std::shared_ptr<Shape> CreateSphereShape(const Transform *o2w,
    const Transform *w2o, bool reverseOrientation,
    const ParamSet &params) {
    Float radius = params.FindOneFloat("radius", 1.f);
    Float zmin = params.FindOneFloat("zmin", -radius);
    Float zmax = params.FindOneFloat("zmax", radius);
    Float phimax = params.FindOneFloat("phimax", 360.f);
    return std::make_shared<Sphere>(o2w, w2o, reverseOrientation, radius,
        zmin, zmax, phimax);
}

```

The appropriate named parameter values are extracted from the parameter list, sensible defaults are used for ones not present, and the appropriate values are passed to the `Sphere` constructor. As another alternative, we could have written the `Sphere` constructor to just take a `ParamSet` as a parameter and extracted the parameters there. We followed this approach instead in order to make it easier to create spheres for other uses without having to create a full `ParamSet` to specify the parameters to it.

Thus, adding a new implementation to `pbrt` requires adding the new source files to the build process so they are compiled and linked to the executable, modifying the appropriate creation function (`MakeShapes()`, `MakeLight()`, etc.) in `core/api.cpp` to look for the new type's name and call its creation function, in addition to implementing the creation function (like `CreateSphereShape()`) that extracts parameters and calls the object's constructor, returning a new instance of the object.

FURTHER READING

The RenderMan® API is described in a number of books (Upstill 1989; Apodaca and Gritz 2000) and in its specification document, which is available online (Pixar Animation Studios 2000). A different approach to rendering APIs is taken by the *mental ray* rendering system, which exposes much more internal information about the system’s implementation to users, allowing it to be extensible in more ways than is possible in other systems. See Driemeyer and Herken (2002) for further information about it.

EXERCISES

- ➊ B.1 One approach for reducing renderer startup time is to support a binary representation of internal data structures that can be written to disk. For example, for complex scenes, creating the ray acceleration aggregates may take more time than the initial parsing of the scene file. An alternative is to modify the system to have the capability of dumping out a representation of the acceleration structure and all of the primitives inside it after it is first created. The resulting file could then be subsequently read back into memory much more quickly than rebuilding the data structure from scratch. However, because C++ doesn’t have native support for saving arbitrary objects to disk and then reading them back during a subsequent execution of the program (a capability known as *serialization* or *pickling* in other languages), adding this feature effectively requires extending many of the objects in pbrt to support this capability on their own.

One additional advantage of this approach is that substantial amounts of computation can be invested in creating high-quality acceleration structures, with the knowledge that this cost doesn’t need to be paid each time the scene is loaded into memory. Implement support for serializing the scene representation and then reusing it across multiple renderings of the scene. How is pbrt’s start-up time (up until when rendering begins) affected? What about overall rendering time?

- ➋ B.2 The material assigned to object instances in pbrt is the current material when the instance was defined inside the `pbrtObjectBegin()`/`pbrtObjectEnd()` block. This can be inconvenient if the user wants to use the same geometry multiple times in a scene, giving it a different material. Fix the API and the implementation of the `TransformedPrimitive` class so that the material assigned to instances is the current material when the instance is instantiated, or, if the user hasn’t set a material, the current material when the instance was created.
- ➌ B.3 Generalize pbrt’s API for specifying animation; the current implementation only allows the user to provide two transformation matrices, only at the start and end of a fixed time range. For specifying more complex motion, a more flexible approach may be useful. One improvement is to allow the user to specify an arbitrary number of *keyframe* transformations, each associated with an arbitrary time.

`pbrtObjectBegin()` 1128

`pbrtObjectEnd()` 1128

`TransformedPrimitive` 252

More generally, the system could be extended to support transformations that are explicit functions of time. For example, a rotation could be described with an expression of the form `Rotate (time * 2 + 1) 0 0 1` to describe a time-varying rotation about the z axis. Extend pbrt to support a more general matrix animation scheme, and render images showing results that aren't possible with the current implementation. What is the performance impact of your changes for scenes with animated objects that don't need the generality of your improvements?

- ③ **B.4** Extend pbrt's API to have some retained mode semantics so that animated sequences of images can be rendered without needing to respecify the entire scene for each frame. Make sure that it is possible to remove some objects from the scene, add others, modify objects' materials and transformations from frame to frame, and so on.
- ② **B.5** In the current implementation, a unique `TransformedPrimitive` is created for each `Shape` with an animated transformation. If many shapes have exactly the same animated transformation, this turns out to be a poor choice. Consider the difference between a million-triangle mesh with an animated transformation versus a million independent triangles, all of which happen to have the same animated transformation.

In the first case, all of the triangles in the mesh are stored in a single instance of a `TransformedPrimitive` with an animated transformation. If a ray intersects the conservative bounding box that encompasses all of the object's motion over the frame time, then it is transformed to the mesh's object space according to the interpolated transformation at the ray's time. At this point, the intersection computation is no different from the intersection test with a static primitive; the only overhead due to the animation is from the larger bounding box and rays that hit the bounding box but not the animated primitive and the extra computation for matrix interpolation and transforming each ray once, according to its time.

In the second case, each triangle is stored in its own `TransformedPrimitive`, all of which happen to have the same `AnimatedTransform`. Each instance of `TransformedPrimitive` will have a large bounding box to encompass each triangle's motion, giving the acceleration structure a difficult set of inputs to deal with: many primitives with substantially overlapping bounding boxes. The impact on ray-primitive intersection efficiency will be high: the ray will be redundantly transformed many times by what happens to be the same recomputed interpolated transformation, and many intersection tests will be performed due to the large bounding boxes. Overall performance will be much worse than the first case.

To address this case, modify the code that implements the pbrt API calls so that if independent shapes are provided with the same animated transformation, they're all collected into a single acceleration structure with a single animated transformation. What is the performance improvement for the worst case outlined above? Is there an impact for more typical scenes with animated primitives?

`AnimatedTransform` 103

`Shape` 123

`TransformedPrimitive` 252

This page intentionally left blank

INDEX OF FRAGMENTS

Bold numbers indicate the first page of a fragment definition, ***bold italic*** numbers indicate an extension of the definition, and roman numbers indicate a use of the fragment.

- ⟨AAMethod Declaration⟩, 643
- ⟨Accept or reject the proposal⟩, 1041, **1042**
- ⟨Account for contributions of clamped octaves in turbulence⟩, 660, 660
- ⟨Account for light contributions along sampled direction w̄⟩, 860, **861**
- ⟨Account for non-symmetry with transmission to different medium⟩, 529, 817, **961**
- ⟨Account for subsurface scattering, if applicable⟩, 877, **915**
- ⟨Account for the direct subsurface scattering component⟩, 915, 915
- ⟨Account for the indirect subsurface scattering component⟩, 915, **916**
- ⟨Accumulate beam transmittance for ray segment⟩, 687, 687
- ⟨Accumulate chain of intersections along ray⟩, 910, **911**
- ⟨Accumulate contribution of jth light to L⟩, 854, **855**
- ⟨Accumulate direct illumination at SPPM camera ray intersection⟩, 978, 978
- ⟨Accumulate light contributions for ray with no intersection⟩, 977, 977
- ⟨Accumulate weighted sums of nearby a_k coefficients⟩, 557, **557**
- ⟨Add camera ray’s contribution to image⟩, 30, **32**
- ⟨Add contribution from dipole for depth z_r to Ed⟩, 929, **930**
- ⟨Add contribution of (a, b) to a_k values⟩, 557, **558**
- ⟨Add contribution of each light source⟩, 34, **36**
- ⟨Add contribution of single scattering at depth t⟩, 934, **934**
- ⟨Add contributions of constant segments before/after samples⟩, 321, **321**
- ⟨Add current block to usedBlocks list⟩, 1074, **1075**
- ⟨Add emitted light at path vertex or from the environment⟩, 877, **877**
- ⟨Add entry to treeletsToBuild for this treelet⟩, 276, **277**
- ⟨Add light contribution from material sampling⟩, 861, **861**
- ⟨Add light’s contribution to reflected radiance⟩, 858, **860**
- ⟨Add photon contribution to nearby visible points⟩, 986, **986**
- ⟨Add photon contribution to visible points in grid[h]⟩, 986, **986**
- ⟨Add pixel’s visible point to applicable grid cells⟩, 981, **982**
- ⟨Add prims and areaLights to scene or current instance⟩, 1124, **1127**
- ⟨Add splat value at pixel⟩, 495, **496**
- ⟨Add the current summand and update the cosine iterates⟩, 559, **560**
- ⟨Add the next series term to F and f⟩, 829, **830**
- ⟨Add visible point to grid cell (x, y, z)⟩, 982, **982**
- ⟨Add visible points to SPPM grid⟩, 979, **981**
- ⟨Adjust normal based on orientation and handedness⟩, 117, **119**
- ⟨Advance to first relevant wavelength segment⟩, 321, **322**
- ⟨Advance to next child node, possibly enqueue other child⟩, 299, **301**
- ⟨Advance to next subtree level if there’s no LBVH split for this bit⟩, 278, **279**
- ⟨Advance v to next higher float⟩, 212, **213**
- ⟨Aggregate Declarations⟩, 255
- ⟨Allocate BucketInfo for SAH partition buckets⟩, 265, **266**
- ⟨Allocate film image storage⟩, 484, **486**
- ⟨Allocate grid for SPPM visible points⟩, 979, **980**
- ⟨Allocate LoopSubdiv vertices and faces⟩, 184, **185**
- ⟨Allocate MemoryArena for tile⟩, 28, **29**
- ⟨Allocate next level of children in mesh tree⟩, 194, **194**
- ⟨Allocate space in perms for radical inverse permutations⟩, 449, **449**

- (Allocate storage to accumulate ak coefficients)*, 557, 557
(Allocate working memory for kd-tree construction), 289, 292, 297
(AnimatedTransform Method Definitions), 103, 104, 106, 108, 110
(AnimatedTransform Private Data), 103, 110
(AnimatedTransform Public Methods), 107
(API Cleanup), 1109, 1111, 1114
(API Forward Declarations), 1125
(API Function Declarations), 1113, 1116, 1117, 1119, 1122
(API Global Variables), 1109
(API Initialization), 1109, 1110, 1114, 1118
(API Local Classes), 1112, 1112, 1114, 1118
(API Macros), 1111, 1112, 1114
(API Static Data), 1110, 1111, 1113, 1114, 1119, 1125
(Append admissible intersection to IntersectionChain), 911, 911
(Apply boundary rule for even vertex), 195, 197
(Apply box filter to checkerboard region), 644, 646
(Apply concentric mapping to point), 778, 779
(Apply edge rules to compute new vertex position), 199, 200
(Apply nSmall small step mutations), 1033, 1033
(Apply one-ring rule for even vertex), 195, 196
(Apply remaining sequence of mutations to sample), 1032, 1033
(Apply shear transformation to translated vertex positions), 158, 159
(Apply sWeights to zoom in s direction), 627, 629
(Approximate derivatives using finite differences), 823, 823, 937
(AreaLight Interface), 734, 734
(Atomically add node to the start of grid[h]'s linked list), 982, 983
(AtomicFloat Private Data), 1086, 1086
(AtomicFloat Public Methods), 1086, 1086, 1087
(Attempt to create the next subpath vertex in path), 1006, 1006
(Bail out if we found a hit closer than the current node), 299, 299
(BDPT Declarations), 992, 996
(BDPT Helper Definitions), 996, 1017, 1021
(BDPT Method Definitions), 1009
(BDPT Utility Functions), 963, 1003, 1004, 1005, 1011, 1016
(BDPTIntegrator Private Data), 992, 993
(BDPTIntegrator Public Methods), 992, 994
(BeckmannDistribution Private Data), 538, 539
(BilerpTexture Declarations), 617
(BilerpTexture Private Data), 617, 617
(BilerpTexture Public Methods), 617, 617, 618
(Bisect range based on value of pred at middle), 1065, 1065
(BlockedArray Private Data), 1076, 1078
(BlockedArray Public Methods), 1076, 1077, 1078, 1079
(Bound image pixels that samples in sampleBounds contribute to), 488, 489
(BoundEdge Public Methods), 292, 292
(Bounds Declarations), 76, 76
(Bounds3 Public Data), 76, 77
(Bounds3 Public Methods), 76, 76, 77, 78, 80, 81
(BoxFilter Declarations), 477
(BoxFilter Method Definitions), 477
(BSDF Declarations), 513, 554, 572
(BSDF Inline Functions), 510, 510, 511, 512, 526, 531, 807
(BSDF Method Definitions), 575, 832
(BSDF Private Data), 572, 573, 573, 583
(BSDF Private Methods), 572, 576
(BSDF Public Data), 572, 573
(BSDF Public Methods), 572, 573, 573, 574, 575, 834
(BSSRDF Declarations), 692, 693, 696, 697, 906
(BSSRDF Interface), 692, 693, 904
(BSSRDF Method Definitions), 699, 905, 908, 912, 914
(BSSRDF Protected Data), 692, 692
(BSSRDF Public Methods), 692, 692
(BSSRDF Utility Declarations), 695
(BSSRDF Utility Functions), 928, 934, 935, 938
(BSSRDFTable Public Data), 697, 698, 698, 914
(BSSRDFTable Public Methods), 697, 698, 700
(Build BVH from primitives), 256, 257
(Build BVH tree for primitives using primitiveInfo), 257, 258
(Build kd-tree for accelerator), 285, 289
(BVHAccel Local Declarations), 257, 258, 271, 275, 281
(BVHAccel Method Definitions), 256, 259, 271, 278, 282
(BVHAccel Private Data), 257, 281
(BVHAccel Private Methods), 280
(BVHAccel Public Types), 256
(BVHAccel Utility Functions), 272, 273, 274
(BVHBuildNode Public Methods), 258, 258, 259
(BxDF Declarations), 513, 515, 521, 522, 523, 524, 528, 531, 532, 547, 548, 550, 555
(BxDF Interface), 513, 513, 513, 514, 515
(BxDF Method Definitions), 515, 522, 523, 525, 529, 533, 536, 547, 550, 551, 556, 563, 806, 807, 811, 813, 814, 815, 816, 819, 821, 831
(BxDF Public Data), 513, 513
(BxDF Utility Functions), 519
(Camera Declarations), 356, 357, 358
(Camera Interface), 356, 356, 357, 949, 953, 954
(Camera Method Definitions), 357
(Camera Public Data), 356, 356
(Cancel marginal PDF factor from tabulated BSSRDF profile), 699, 700, 914
(Capture escaped rays when tracing from the camera), 1007, 1020
(Check for intersections inside leaf node), 299, 301
(Check for ray intersection against x and y slabs), 129, 129
(Check one primitive inside leaf node), 301, 301
(Check quadric shape t0 and t1 for nearest intersection), 136, 136
(Check ray against BVH node), 283, 283
(Checkerboard2DTexture Private Data), 642, 643
(Checkerboard2DTexture Public Methods), 642, 643, 643
(Checkerboard3DTexture Private Data), 647, 648
(Checkerboard3DTexture Public Methods), 647, 647, 648
(CheckerboardTexture Declarations), 642, 647
(Choose albedo values of the diffusion profile discretization), 935, 936

- (Choose level of detail for EWA lookup and perform EWA filtering), 635, 636
- (Choose light to shoot photon from), 984, 985
- (Choose point on disk oriented toward infinite light direction), 958, 959
- (Choose projection axis for BSSRDF sampling), 908, 908
- (Choose radius values of the diffusion profile discretization), 935, 935
- (Choose spectral channel for BSSRDF sampling), 908, 909
- (Choose split axis position for interior node), 290, 293
- (Choose two dimensions to use for ray offset computation), 603, 604
- (Choose which axis to split along), 293, 293
- (Choose which BxDF to sample), 832, 833
- (Clamp ellipse eccentricity if too large), 635, 636
- (Classify primitives with respect to split), 290, 296
- (Clean up after rendering), 1129, 1130
- (CoefficientSpectrum Protected Data), 316, 316
- (CoefficientSpectrum Public Data), 316, 318
- (CoefficientSpectrum Public Methods), 316, 316, 316, 317, 318
- (Compute (u, v) offsets at auxiliary points), 601, 603
- (Compute 1D array samples for GlobalSampler), 430, 431
- (Compute absolute error for transformed point), 228, 229
- (Compute acceptance probability for proposed sample), 1041, 1041
- (Compute alpha for direction w), 543, 543
- (Compute and return RGB colors for tabulated BSDF), 558, 559
- (Compute angle α from center of sphere to sampled point on surface), 841, 844
- (Compute arrayEndDim for dimensions used for array samples), 430, 431
- (Compute auxiliary intersection points with plane), 601, 602
- (Compute average value of given SPD over i th sample's range), 319, 320
- (Compute barycentric coordinates and t value for triangle intersection), 158, 163
- (Compute base-2 radical inverse), 444, 446
- (Compute bound of primitive centroids, choose split dimension dim), 259, 261
- (Compute bounding box of all primitive centroids), 271, 271
- (Compute bounding box of curve segment, curveBounds), 174, 174
- (Compute bounding box of projection of rear element on sampling plane), 392, 392
- (Compute bounding box of ray, rayBounds), 174, 174
- (Compute bounds for kd-tree construction), 289, 289
- (Compute bounds of all primitives in BVH node), 259, 260
- (Compute BSDF at photon intersection point), 987, 988
- (Compute BSDF at SPPM camera ray intersection), 978, 978
- (Compute BSDF or phase function's value for light sample), 858, 859
- (Compute BSSRDF profile bounds and intersection height), 908, 909
- (Compute BSSRDF profile radius under projection along each axis), 912, 913
- (Compute BSSRDF sampling ray segment), 908, 910
- (Compute bump-mapped differential geometry), 589, 590
- (Compute cardinal points for film side of lens system), 387, 387
- (Compute cardinal points for scene side of lens system), 387, 388
- (Compute cell indices for dots), 654, 654
- (Compute child surface areas for split at edgeT), 294, 295
- (Compute closed-form box-filtered Checkerboard2DTexture value), 643, 644
- (Compute coefficients for fundamental forms), 139, 140, 146
- (Compute conditional sampling distribution for \tilde{v}), 785, 785
- (Compute coordinate system for sphere sampling), 841, 841
- (Compute $\cos \theta$ for Henyey–Greenstein sample), 899, 899
- (Compute $\cos \theta_t$ using Snell's law), 531, 531
- (Compute cosine of cone surrounding projection directions), 726, 727
- (Compute cosine term of Oren–Nayar model), 536, 536
- (Compute cost for split at i th edge), 294, 295
- (Compute cost of all splits for axis to find best), 293, 294
- (Compute cosThetaT using Snell's law), 519, 520
- (Compute costs for splitting after each bucket), 265, 267
- (Compute curve $\partial p / \partial v$ for flat and cylinder curves), 180, 181
- (Compute cylinder $\partial n / \partial u$ and $\partial n / \partial v$), 145, 146
- (Compute cylinder $\partial p / \partial u$ and $\partial p / \partial v$), 145, 145
- (Compute cylinder hit point and ϕ), 144, 145, 145
- (Compute deltas for triangle partial derivatives), 164, 164
- (Compute δ_e term for triangle t error bounds), 234, 235
- (Compute δ_t term for triangle t error bounds and check t), 234, 235
- (Compute δ_x and δ_y terms for triangle t error bounds), 234, 234
- (Compute δ_z term for triangle t error bounds), 234, 234
- (Compute differential changes in origin for orthographic camera rays), 361, 363
- (Compute differential changes in origin for perspective camera rays), 365, 367
- (Compute differential reflected directions), 607, 608
- (Compute dipole fluence rate $\phi_D(r)$ using Equation (15.25)), 929, 930
- (Compute dipole vector irradiance $-n \cdot E_D(r)$ using Equation (15.27)), 929, 930
- (Compute direct lighting for DirectLightingIntegrator integrator), 853
- (Compute direction w_i for Henyey–Greenstein sample), 899, 899
- (Compute directional density depending on the vertex type), 1001, 1001
- (Compute directions to preceding and next vertex), 1001, 1001
- (Compute $\partial n / \partial u$ and $\partial n / \partial v$ from fundamental form coefficients), 139, 140, 146
- (Compute $\partial p / \partial u$ and $\partial p / \partial v$ for curve intersection), 180, 180
- (Compute edge function coefficients e_0, e_1 , and e_2), 158, 161
- (Compute effect of visibility for light source sample), 858, 859
- (Compute effective albedo ρ_{eff} and CDF for importance sampling), 936, 937
- (Compute effective transport coefficient σ_{tr} based on D_G), 928, 929

- ⟨Compute ellipse coefficients to bound EWA filter region⟩, 637, 637
 ⟨Compute ellipse minor and major axes⟩, 635, 636
 ⟨Compute emitted and reflected light at ray intersection point⟩, 33, 34
 ⟨Compute emitted light if ray hit an area light source⟩, 34, 35
 ⟨Compute emitted radiance for blackbody at wavelength $\lambda_{\text{lambda}}[i]$ ⟩, 710, 710
 ⟨Compute environment camera ray direction⟩, 377, 377
 ⟨Compute error bounds for cone intersection⟩, 227
 ⟨Compute error bounds for curve intersection⟩, 180, 227
 ⟨Compute error bounds for cylinder intersection⟩, 144, 225
 ⟨Compute error bounds for disk intersection⟩, 148, 225
 ⟨Compute error bounds for intersection computed with ray equation⟩, 227, 227
 ⟨Compute error bounds for sampled point on triangle⟩, 839, 840
 ⟨Compute error bounds for sphere intersection⟩, 134, 225
 ⟨Compute error bounds for triangle intersection⟩, 157, 227
 ⟨Compute exit pupil bounds at sampled points on the film⟩, 379, 390
 ⟨Compute falloff inside spotlight cone⟩, 724, 724
 ⟨Compute film image bounds⟩, 484, 485
 ⟨Compute first node weight w_0 ⟩, 562, 563
 ⟨Compute Fourier coefficients a_k for (μ_i, μ_o) ⟩, 556, 557, 819
 ⟨Compute general sphere PDF⟩, 844, 845
 ⟨Compute gradient weights⟩, 649, 651
 ⟨Compute grid bounds for SPPM visible points⟩, 979, 981
 ⟨Compute hit t and partial derivatives for curve intersection⟩, 176, 180
 ⟨Compute image plane bounds at $z = 1$ for PerspectiveCamera⟩, 365, 951
 ⟨Compute image resampling weights for i-th texel⟩, 628, 628
 ⟨Compute InfiniteAreaLight ray PDFs⟩, 959, 960
 ⟨Compute initial node weights w_1 and w_2 ⟩, 562, 563
 ⟨Compute initial parametric range of ray inside kd-tree extent⟩, 297, 297
 ⟨Compute integral of step function at x_i ⟩, 758, 758
 ⟨Compute interpolated matrix as product of interpolated components⟩, 106, 107
 ⟨Compute intersection of ray with lens element⟩, 383, 383
 ⟨Compute lens area of perspective camera⟩, 950, 953, 953, 955
 ⟨Compute lens-film distance for given focus distance⟩, 379, 389
 ⟨Compute lightDistr for sampling lights proportional to power⟩, 973, 974
 ⟨Compute line w that gives minimum distance to sample point⟩, 176, 178
 ⟨Compute marginal sampling distribution $p[\vec{v}]$ ⟩, 785, 785
 ⟨Compute material parameters and minimum t below the critical angle⟩, 934, 934
 ⟨Compute MIPMap level for trilinear filtering⟩, 632, 633
 ⟨Compute MIS weight for connection strategy⟩, 1009, 1012
 ⟨Compute Morton indices of primitives⟩, 271, 271
 ⟨Compute new odd edge vertices⟩, 194, 198
 ⟨Compute new transformation matrix M without translation⟩, 104, 104
 ⟨Compute next matrix R_{next} in series⟩, 105, 105
 ⟨Compute next sine and cosine iterates⟩, 829, 830
 ⟨Compute noise cell coordinates and offsets⟩, 649, 649
 ⟨Compute non-classical diffusion coefficient D_G using Equation (15.24)⟩, 928, 929
 ⟨Compute norm of difference between R and R_{next} ⟩, 105, 105
 ⟨Compute number of octaves for antialiased FBM⟩, 656, 658, 660
 ⟨Compute number of samples to use for each light⟩, 852, 853
 ⟨Compute number of tiles to use for SPPM camera pass⟩, 975, 976
 ⟨Compute number of tiles, nTiles, to use for parallel rendering⟩, 27, 28
 ⟨Compute object-space control points for curve segment, cpObj⟩, 171, 172, 173
 ⟨Compute odd vertex on kth edge⟩, 198, 199
 ⟨Compute offset along CDF segment⟩, 759, 759
 ⟨Compute offset positions and evaluate displacement texture⟩, 589, 590
 ⟨Compute offset rays for PerspectiveCamera ray differentials⟩, 368
 ⟨Compute overall PDF with all matching BxDFs⟩, 832, 834
 ⟨Compute parametric distance along ray to split plane⟩, 299, 299
 ⟨Compute PDF for importance arriving at ref⟩, 954, 955
 ⟨Compute PDF for sampled infinite light direction⟩, 849, 850
 ⟨Compute PDF for sampled offset⟩, 759, 759
 ⟨Compute PDF of wi for microfacet reflection⟩, 811, 813
 ⟨Compute perfect specular reflection direction⟩, 525, 526, 817
 ⟨Compute planar sampling density for infinite light sources⟩, 1002, 1022
 ⟨Compute plane intersection for disk⟩, 148, 148
 ⟨Compute point on plane of focus⟩, 374, 375
 ⟨Compute projective camera screen transformations⟩, 360, 360
 ⟨Compute projective camera transformations⟩, 360, 360
 ⟨Compute quadratic cylinder coefficients⟩, 144, 144
 ⟨Compute quadratic sphere coefficients⟩, 134, 135, 141
 ⟨Compute quadratic t values⟩, 1079, 1080
 ⟨Compute radiance L for SPPM pixel pixel⟩, 990
 ⟨Compute raster and camera sample positions⟩, 364, 364, 367
 ⟨Compute ray after transformation by PrimitiveToWorld⟩, 253, 253
 ⟨Compute ray differential rd for specular reflection⟩, 38, 607
 ⟨Compute ray differentials for OrthographicCamera⟩, 364, 364
 ⟨Compute ray direction for specular transmission⟩, 529, 529, 817
 ⟨Compute reduced scattering coefficients σ_s' , σ_t' and albedo ρ' ⟩, 928, 929
 ⟨Compute reflectance SampledSpectrum with $\text{rgb}[0]$ as minimum⟩, 330, 331
 ⟨Compute representation of depth-first traversal of BVH tree⟩, 257, 281
 ⟨Compute resolution of SPPM grid in each dimension⟩, 979, 981
 ⟨Compute reverse area density at preceding vertex⟩, 1006, 1008
 ⟨Compute rotation of first basis vector⟩, 91, 91
 ⟨Compute sample bounds for tile⟩, 28, 29
 ⟨Compute sample PDF and return the spatial BSSRDF term S_p ⟩, 908, 912
 ⟨Compute sample values for photon ray leaving light source⟩, 984, 985

- (Compute sample's raster bounds), 490, 490
 (Compute sampling density for non-infinite light sources), 1002, 1002
 (Compute sampling distributions for rows and columns of image), 847, 848
 (Compute scalar-valued image img from environment map), 847, 848
 (Compute scale S using rotation and original matrix), 104, 105
 (Compute scaled hit distance to triangle and test against ray t range), 158, 162
 (Compute scattering functions and skip over medium boundaries), 877, 878
 (Compute scattering functions for mode and skip over medium boundaries), 1007, 1007
 (Compute scattering functions for surface interaction), 34, 35
 (Compute scattering profile for chosen albedo ρ), 936, 937
 (Compute shading bitangent ts for triangle and adjust ss), 166, 167
 (Compute shading normal ns for triangle), 166, 166
 (Compute shading tangent ss for triangle), 166, 166
 (Compute shading.nf for SurfaceInteraction), 119, 119
 (Compute sine and tangent terms of Oren–Nayar model), 536, 537
 (Compute specular reflection direction wi and BSDF value), 37, 38
 (Compute specular reflection for FresnelSpecular), 816, 817
 (Compute specular transmission for FresnelSpecular), 816, 817
 (Compute sphere $\partial n/\partial u$ and $\partial n/\partial v$), 137, 139
 (Compute sphere $\partial p/\partial u$ and $\partial p/\partial v$), 137, 138
 (Compute sphere hit position and ϕ), 134, 137, 137, 141
 (Compute spline weights to interpolate BSSRDF on channel ch), 699, 699
 (Compute squared radius and filter texel if inside ellipse), 639, 639
 (Compute starting index in output array for each bucket), 274, 275
 (Compute stride in samples for visiting each pixel area), 451, 453
 (Compute sum of octaves of noise for FBm), 656, 658
 (Compute sum of octaves of noise for turbulence), 660, 660
 (Compute surface normal and sampled point on sphere), 841, 844
 (Compute surface normal for sampled point on triangle), 839, 840
 (Compute tangents of boundary face), 203, 206
 (Compute tangents of interior face), 203, 204
 (Compute $\tan^2 \theta$ and ϕ for Beckmann distribution sample), 808, 809
 (Compute texel (s, t) accounting for boundary conditions), 631, 631
 (Compute texel (s, t) in s-zoomed image), 629, 629
 (Compute texture coordinate differentials for sphere (u, v) mapping), 611, 612
 (Compute texture differentials for 2D (u, v) mapping), 610, 611
 (Compute the diffusion profile for the i-th albedo sample), 935, 936
 (Compute the discrete probability of sampling light, pdfChoice), 1003, 1003
 (Compute the effective standard deviation and apply perturbation to X_i), 1033, 1034
 (Compute the ellipse's (s, t) bounding box in texture space), 637, 638
 (Compute the scattered direction for FourierBSDF), 819, 820
 (Compute the t parameter and powers), 562, 562
 (Compute the transmittance and sampling density), 893, 894
 (Compute θ and ϕ values for sample in cone), 841, 842
 (Compute translation of lens, delta, to focus at focusDistance), 388, 389
 (Compute triangle partial derivatives), 157, 164
 (Compute trilinear interpolation of weights), 649, 653
 (Compute u coordinate of curve intersection point and hitWidth), 176, 179
 (Compute v coordinate of curve intersection point), 176, 180
 (Compute valence of boundary vertex), 191, 191
 (Compute valence of interior vertex), 191, 191
 (Compute value of BSDF for sampled direction), 832, 834
 (Compute vertex tangents on limit surface), 193, 203
 (Compute voxel coordinates and offsets for p), 691, 691
 (Compute weights and original BxDFs for mix material), 582, 582
 (Compute ω_h from ω_o and ω_i for microfacet transmission), 814, 814
 (Compute XYZ matching functions for SampledSpectrum), 324, 324
 (Compute $[t_{\min}, t_{\max}]$ interval of ray's overlap with medium bounds), 896, 896, 898
 (Consider hypothetical connection strategies along the camera subpath), 1016, 1017
 (Consider hypothetical connection strategies along the light subpath), 1016, 1017
 (ConstantTexture Declarations), 615
 (ConstantTexture Public Methods), 615, 615
 (Convert EWA coordinates to appropriate scale for level), 637, 637
 (Convert image to RGB and compute final pixel values), 494, 495
 (Convert infinite light sample point to direction), 849, 849
 (Convert light sample weight to solid angle measure), 837, 838
 (Convert pixel XYZ color to RGB), 495, 495
 (Convert r into unitless optical radius r_{optical}), 699, 699, 914
 (Convert reflectance spectrum to RGB), 330, 330
 (Convert texels to type Tmemory and create MIPMap), 620, 621
 (Copy final result from tempVector, if needed), 274, 275
 (Correct subpath sampling densities for infinite area lights), 1005, 1021
 (Cosine-sample the hemisphere, flipping the direction if necessary), 806, 806, 814
 (Count number of zero bits in array for current radix sort bit), 274, 274
 (Create aggregate for instance Primitives), 1128, 1129
 (Create and enqueue ParallelForLoop for this loop), 1088, 1089
 (Create and initialize new odd vertex), 199, 199
 (Create and return interior LBVH node), 278, 279

- ⟨Create and return leaf node of LBVH treelet⟩, 278, 278
 ⟨Create and return SAH BVH from LBVH treelets⟩, 271, 280
 ⟨Create animatedInstanceToWorld transform for instance⟩, 1128, 1129
 ⟨Create Float texture and store in floatTextures⟩, 1121, 1122
 ⟨Create GeometricPrimitive(s) for animated shape⟩, 1126, 1126
 ⟨Create grid of all SPPM visible points⟩, 976, 979
 ⟨Create initial shape or shapes for animated shape⟩, 1126, 1126
 ⟨Create interior flattened BVH node⟩, 282, 282
 ⟨Create LBVH treelets at bottom of BVH⟩, 271, 275
 ⟨Create LBVs for treelets in parallel⟩, 275, 277
 ⟨Create leaf BVHBuildNode⟩, 259, 260, 261, 268
 ⟨Create leaf if no good splits were found⟩, 290, 296
 ⟨Create microfacet distribution distrib for plastic material⟩, 581, 581
 ⟨Create MIPMap for filename⟩, 620, 620
 ⟨Create mipmap for GoniophotometricLight⟩, 730, 730
 ⟨Create one-valued MIPMap⟩, 620, 622
 ⟨Create ProjectionLight MIP map⟩, 726, 726
 ⟨Create scene and render⟩, 1129, 1130
 ⟨Create shapes for shape name⟩, 1124, 1124
 ⟨Create single TransformedPrimitive for prims⟩, 1126, 1126
 ⟨Curve Declarations⟩, 168
 ⟨Curve Method Definitions⟩, 171, 171, 173, 174
 ⟨Curve Private Data⟩, 168, 170
 ⟨Curve Public Methods⟩, 168, 170
 ⟨Curve Utility Functions⟩, 172, 175, 180
 ⟨CurveCommon Declarations⟩, 171
 ⟨CurveType Declarations⟩, 169
 ⟨Cylinder Declarations⟩, 142
 ⟨Cylinder Method Definitions⟩, 143, 144, 146, 839
 ⟨Cylinder Private Data⟩, 142, 143
 ⟨Cylinder Public Methods⟩, 142, 143
 ⟨CylindricalMapping2D Private Methods⟩, 612, 613
 ⟨Declare IntersectionChain and linked list⟩, 910, 910
 ⟨Declare variables for forward and reverse probability densities⟩, 1005, 1006
- ⟨Define a lambda function to interpolate table entries⟩, 824, 825
 ⟨Define CVanDerCorput Generator Matrix⟩, 463, 464
 ⟨Define helper function remap0 that deals with Dirac delta functions⟩, 1016, 1016
 ⟨Determine exitance scale factors using Equations (15.31) and (15.32)⟩, 928, 929
 ⟨Determine length d of connecting segment and $\cos \theta_o$ ⟩, 934, 934
 ⟨Determine linear extrapolation distance z_e using Equation (15.28)⟩, 928, 929
 ⟨Determine offset and coefficients for the alpha parameter⟩, 824, 824
 ⟨Determine offsets and weights for μ_i and μ_o ⟩, 557, 557
 ⟨Determine the number of available strategies and pick a specific one⟩, 1036, 1036
 ⟨DiffuseAreaLight Declarations⟩, 736
 ⟨DiffuseAreaLight Method Definitions⟩, 736, 736, 845, 957
 ⟨DiffuseAreaLight Protected Data⟩, 736, 736
 ⟨DiffuseAreaLight Public Methods⟩, 736, 736
 ⟨DirectLightingIntegrator Declarations⟩, 851
 ⟨DirectLightingIntegrator Method Definitions⟩, 852
 ⟨DirectLightingIntegrator Private Data⟩, 851, 852, 853
 ⟨Discard directions behind projection light⟩, 727, 727
 ⟨Disk Declarations⟩, 146
 ⟨Disk Method Definitions⟩, 148, 148, 150, 838
 ⟨Disk Private Data⟩, 146, 147
 ⟨Disk Public Methods⟩, 146, 146
 ⟨DistantLight Declarations⟩, 731
 ⟨DistantLight Method Definitions⟩, 731, 732, 733, 958
 ⟨DistantLight Private Data⟩, 731, 731, 732
 ⟨DistantLight Public Methods⟩, 731, 732
 ⟨Distribution1D Public Data⟩, 758, 758
 ⟨Distribution1D Public Methods⟩, 758, 758, 759, 760
 ⟨Distribution2D Private Data⟩, 785, 785, 785
 ⟨Distribution2D Public Methods⟩, 785, 786, 787
 ⟨DotsTexture Declarations⟩, 653
 ⟨DotsTexture Private Data⟩, 653, 653
- ⟨DotsTexture Public Methods⟩, 653, 653, 654
 ⟨EFloat Inline Functions⟩, 220
 ⟨EFloat Private Data⟩, 218, 219
 ⟨EFloat Public Methods⟩, 218, 219, 220
 ⟨Either create leaf or split primitives at selected SAH bucket⟩, 265, 268
 ⟨EndpointInteraction Declarations⟩, 996
 ⟨EndpointInteraction Public Methods⟩, 996, 997, 997
 ⟨Enlarge MLTSampler::X if necessary and get current X_i ⟩, 1032, 1032
 ⟨Enqueue secondChild in todo list⟩, 301, 301
 ⟨Ensure correct orientation of the geometric normal⟩, 165, 166
 ⟨Ensure that computed triangle t is conservatively greater than zero⟩, 158, 234
 ⟨Ensure there are no pushed graphics states⟩, 1129, 1129
 ⟨EnvironmentCamera Declarations⟩, 376
 ⟨EnvironmentCamera Method Definitions⟩, 377
 ⟨EnvironmentCamera Public Methods⟩, 376, 376
 ⟨Erase primitives and lights from RenderOptions⟩, 1130, 1130
 ⟨Estimate direct lighting using sample arrays⟩, 855, 855
 ⟨Estimate one term of ρ_{hd} ⟩, 831, 831
 ⟨Estimate one term of ρ_{hh} ⟩, 831, 832
 ⟨Estimate screen space change in p and (u, v) ⟩, 601, 601
 ⟨Evaluate BSDF for light sampling strategy⟩, 859, 859
 ⟨Evaluate dipole integrand E_d at z_r and add to E_d ⟩, 928, 929
 ⟨Evaluate $F(\phi)$ and its derivative $f(\phi)$ ⟩, 828, 829
 ⟨Evaluate filter value at (x, y) pixel⟩, 491, 492
 ⟨Evaluate Fourier expansion for angle ϕ ⟩, 556, 558
 ⟨Evaluate motion derivative in interval form, return if no zeros⟩, 113, 113
 ⟨Evaluate phase function for light sampling strategy⟩, 859, 900
 ⟨Evaluate probability of sampling w_i ⟩, 821, 821
 ⟨Evaluate radiance along camera ray⟩, 30, 31
 ⟨Evaluate single check if filter is entirely inside one of them⟩, 644, 645
 ⟨Evaluate single-scattering integrand and add to E_{ss} ⟩, 934, 934

- (Evaluate target function and its derivative in Horner form), 825, 827
 (Evaluate textures for MatteMaterial material and allocate BRDF), 579, 579
 (Execute all BDPT connection strategies), 994, 995
 (Execute connection strategy and return the radiance estimate), 1036, 1037
 (Execute the (s, t) connection strategy and update L), 995, 995
 (Expand bounding box for any motion derivative zeros found), 110, 111
 (Expand bounds to account for sample spacing), 391, 393
 (Expand pupil bounds if ray makes it through the lens system), 392, 392
 (Express $p_i - p_o$ and n_i with respect to local coordinates at p_o), 912, 912
 (Extract rotation R from transformation matrix), 104, 105
 (Extract translation T from transformation matrix), 104, 104
 (Fall back to a bisection step when t is out of bounds), 825, 826
 (FBmTexture Declarations), 660
 (FBmTexture Public Methods), 660, 662, 662
 (Figure out which η is incident and which is transmitted), 529, 529, 817
 (Fill in SurfaceInteraction from triangle hit), 157, 165
 (Film Declarations), 484, 489
 (Film Method Definitions), 484, 487, 488, 493, 494
 (Film Private Data), 484, 486, 487, 493, 496
 (Film Private Methods), 484, 493
 (Film Public Data), 484, 485, 485
 (FilmTile Private Data), 489, 489
 (FilmTile Public Methods), 489, 489, 490, 492, 493
 (FilmTilePixel Declarations), 489
 (Filter Declarations), 474
 (Filter four texels from finer level of pyramid), 631, 632
 (Filter Interface), 474, 474, 475
 (Filter Public Data), 474, 475
 (Find (u, v) sample coordinates in infinite light texture), 849, 849
 (Find any motion derivative zeros for the component c), 110, 111
 (Find bucket to split at that minimizes SAH metric), 265, 267
 (Find camera ray after shifting one pixel in the x direction), 357, 358
 (Find closest ray intersection or return background radiance), 33, 34
 (Find exit pupil bound for sample distance from film center), 393, 393
 (Find height x from optical axis for parallel rays), 387, 387
 (Find intersection and compute transmittance), 861, 861
 (Find intervals of primitives for each treelet), 275, 276
 (Find LBVH split point for this dimension), 278, 279
 (Find location of sample points on x segment and rear lens element), 392, 392
 (Find next path vertex and accumulate contribution), 876, 877
 (Find parametric representation of cylinder hit), 144, 145
 (Find parametric representation of disk hit), 148, 149
 (Find parametric representation of sphere hit), 134, 137
 (Find point on film, pFilm, corresponding to sample.pFilm), 394, 394
 (Find quadratic discriminant), 1079, 1079
 (Find radical inverse base scales and exponents that cover sampling area), 451, 452
 (Find surrounding CDF segments and offset), 759, 759, 760
 (Find the set of loop iterations to run next), 1091, 1091
 (Find the zenith angle cosines and azimuth difference angle), 556, 556, 821
 (Finish initialization of RealisticCamera ray), 394, 395
 (Finish vertex initialization), 184, 190
 (Flip R[1] if needed to select shortest path), 103, 106
 (floatfile.h*), 1070
 (Follow camera paths for tile in image for SPPM), 976, 976
 (Follow camera ray path until a visible point is created), 976, 977
 (Follow ith Markov chain for nChainMutations), 1039, 1040
 (Follow photon path for photonIndex), 983, 984
 (Follow photon path through scene and record intersections), 984, 986
 (Follow ray through BVH nodes to find primitive intersections), 282, 283
 (Fourier Interpolation Definitions), 559, 828
 (FourierBSDF Private Data), 555, 556
 (FourierBSDF Public Methods), 555, 556
 (FourierBSDFTable Public Data), 554, 554, 554, 555, 818, 819, 820
 (FourierBSDFTable Public Methods), 554, 554, 555
 (FourierMaterial Declarations), 583
 (FourierMaterial Method Definitions), 583, 584
 (FourierMaterial Private Data), 583, 583
 (Free MemoryArena memory from computing image sample value), 30, 32
 (Fresnel Interface), 521, 522
 (FresnelBlend Private Data), 550, 550
 (FresnelBlend Public Methods), 550, 551
 (FresnelConductor Public Methods), 522, 522
 (FresnelDielectric Public Methods), 522, 522
 (FresnelSpecular Private Data), 531, 532
 (FresnelSpecular Public Methods), 531, 532
 (Function Definitions), 2, 3
 (GaussianFilter Declarations), 478
 (GaussianFilter Method Definitions), 479
 (GaussianFilter Private Data), 478, 478
 (GaussianFilter Public Methods), 478, 478
 (GaussianFilter Utility Functions), 478, 479
 (General pbRT Initialization), 324, 1109
 (Generate 1D and 2D array samples using (0, 2)-sequence), 463, 463
 (Generate 1D and 2D pixel sample components using (0, 2)-sequence), 463, 463
 (Generate a camera subpath with exactly t vertices), 1036, 1037
 (Generate a light subpath with exactly s vertices), 1036, 1037
 (Generate a single sample using BDPT), 994
 (Generate arrays of stratified samples for the pixel), 434, 440
 (Generate bootstrap samples and compute normalization constant b), 1038, 1038
 (Generate bsdfSample for outgoing photon sample), 988, 988
 (Generate camera ray for current sample), 30, 31
 (Generate camera ray for pixel for SPPM), 976, 977
 (Generate first vertex on camera subpath and start random walk), 1003, 1004
 (Generate first vertex on light subpath and start random walk), 1004, 1005

- ⟨Generate *i*th bootstrap sample⟩, 1038, 1039
 ⟨Generate *i*th LBVH treelet⟩, 277, 277
 ⟨Generate LHS samples along diagonal⟩, 440, 440
 ⟨Generate next ray segment or return final transmittance⟩, 718, 719
 ⟨Generate photonRay from light source and initialize beta⟩, 984, 985
 ⟨Generate random digit permutations for Halton sampler⟩, 451, 452
 ⟨Generate random permutation for *i*th prime base⟩, 449, 449
 ⟨Generate sample point inside exit pupil bound⟩, 393, 393
 ⟨Generate single stratified samples for the pixel⟩, 434, 437
 ⟨Generate SPPM visible points⟩, 975, 976
 ⟨GeometricPrimitive Declarations⟩, 250
 ⟨GeometricPrimitive Method Definitions⟩, 251, 251
 ⟨GeometricPrimitive Private Data⟩, 250, 250
 ⟨Geometry Inline Functions⟩, 62, 63, 64, 65, 66, 67, 70, 71, 72, 78, 79, 80, 127, 128, 231, 346
 ⟨Get animatedObjectToWorld transform for shape⟩, 1126, 1127
 ⟨Get BxDF pointer for chosen component⟩, 833, 833
 ⟨Get FilmTile for tile⟩, 28, 30
 ⟨Get new block of memory for MemoryArena⟩, 1074, 1075
 ⟨Get next free node from nodes array⟩, 290, 290
 ⟨Get node children pointers for ray⟩, 299, 300
 ⟨Get one-ring vertices for boundary vertex⟩, 196, 197
 ⟨Get one-ring vertices for interior vertex⟩, 196, 197
 ⟨Get pointer light to the light source at the vertex⟩, 1002, 1002, 1003
 ⟨Get sampler instance for tile⟩, 28, 29
 ⟨Get SPPMPixel for pPixel⟩, 977, 977
 ⟨Get triangle vertices in p0, p1, and p2⟩, 157, 157, 167, 839
 ⟨Get work from workList and run loop iterations⟩, 1092, 1093
 ⟨Global Constants⟩, 210, 214, 232, 1063, 1072
 ⟨Global Forward Declarations⟩, 315, 1062, 1109
 ⟨Global Include Files⟩, 1061, 1068
 ⟨Global Inline Functions⟩, 211, 212, 217, 621, 622, 1062, 1063, 1064, 1065, 1069, 1079
 ⟨Global Macros⟩, 1071
 ⟨GlobalSampler Private Data⟩, 429, 430, 431
 ⟨GlobalSampler Public Methods⟩, 429, 429, 430
 ⟨GonioPhotometricLight Declarations⟩, 728
 ⟨GonioPhotometricLight Method Definitions⟩, 731
 ⟨GonioPhotometricLight Private Data⟩, 728, 730
 ⟨GonioPhotometricLight Public Methods⟩, 728, 730, 730
 ⟨Grab next node to process from todo list⟩, 299, 302
 ⟨Graphics State⟩, 1117, 1118, 1122, 1123
 ⟨Graphics State Methods⟩, 1118, 1125, 1125
 ⟨GridDensityMedium Declarations⟩, 690
 ⟨GridDensityMedium Method Definitions⟩, 691, 896, 898
 ⟨GridDensityMedium Private Data⟩, 690, 690, 896
 ⟨GridDensityMedium Public Methods⟩, 690, 690, 691
 ⟨HaltonSampler Declarations⟩, 450
 ⟨HaltonSampler Local Constants⟩, 453
 ⟨HaltonSampler Method Definitions⟩, 451, 453, 454
 ⟨HaltonSampler Private Data⟩, 450, 452, 453, 454
 ⟨HaltonSampler Private Methods⟩, 450, 452
 ⟨Handle all other bidirectional connection cases⟩, 1009, 1011
 ⟨Handle an interaction with a medium or a surface⟩, 900, 902
 ⟨Handle boundary conditions for matrix interpolation⟩, 106, 106
 ⟨Handle cases with out-of-bounds range or single sample only⟩, 321, 321
 ⟨Handle degeneracy at the origin⟩, 778, 779
 ⟨Handle degenerate cases for microfacet reflection⟩, 547, 547
 ⟨Handle infinity and negative zero for NextFloatUp()⟩, 212, 212
 ⟨Handle new edge⟩, 189, 189
 ⟨Handle opaque surface along ray's path⟩, 718, 718
 ⟨Handle previously seen edge⟩, 189, 189
 ⟨Handle scattering at point in medium for volumetric path tracer⟩, 902, 902
 ⟨Handle sphere mapping discontinuity for coordinate differentials⟩, 611, 612
 ⟨Handle surface interaction for path generation⟩, 1006, 1007
 ⟨Handle total internal reflection⟩, 520, 520
 ⟨Handle total internal reflection for transmission⟩, 531, 531
 ⟨Handle zero determinant for triangle partial derivative matrix⟩, 164, 164
 ⟨Help out with parallel loop iterations in the current thread⟩, 1088, 1091
 ⟨HenyeyGreenstein Declarations⟩, 682
 ⟨HenyeyGreenstein Method Definitions⟩, 682, 899
 ⟨HenyeyGreenstein Public Methods⟩, 682, 682
 ⟨HomogeneousMedium Declarations⟩, 689
 ⟨HomogeneousMedium Method Definitions⟩, 689, 893
 ⟨HomogeneousMedium Private Data⟩, 689, 689
 ⟨HomogeneousMedium Public Methods⟩, 689, 689
 ⟨Ignore invalid connections related to infinite area lights⟩, 1009, 1020
 ⟨ImageIO Declarations⟩, 1067, 1068
 ⟨ImageTexture Declarations⟩, 619
 ⟨ImageTexture Method Definitions⟩, 619, 620
 ⟨ImageTexture Private Data⟩, 619, 619, 620
 ⟨ImageTexture Private Methods⟩, 619, 622, 623
 ⟨ImageTexture Public Methods⟩, 619, 622, 623
 ⟨Importance sample the BSSRDF⟩, 915, 915
 ⟨Importance sample the luminance Fourier expansion⟩, 819, 820
 ⟨InfiniteAreaLight Declarations⟩, 737
 ⟨InfiniteAreaLight Method Definitions⟩, 739, 741, 849, 850, 959
 ⟨InfiniteAreaLight Private Data⟩, 737, 740, 740, 848
 ⟨InfiniteAreaLight Public Methods⟩, 737, 740
 ⟨Initialize A, Bx, and By matrices for offset computation⟩, 603, 604
 ⟨Initialize BucketInfo for SAH partition buckets⟩, 265, 266
 ⟨Initialize CameraSample for current sample⟩, 30, 30
 ⟨Initialize common variables for Whitted integrator⟩, 34, 34
 ⟨Initialize cosine iterates⟩, 559, 559

- ⟨Initialize depth of field parameters⟩, 360, 374
 ⟨Initialize diffuse component of plastic material⟩, 581, 581
 ⟨Initialize dynamically sampled vertex and L for $t = 1$ case⟩, 1010, 1010
 ⟨Initialize edges for axis⟩, 293, 293
 ⟨Initialize EFloat ray coordinate values⟩, 135, 135, 144
 ⟨Initialize EWA filter weights if needed⟩, 625, 639
 ⟨Initialize F and f with the first series term⟩, 829, 830
 ⟨Initialize first three columns of viewing matrix⟩, 92, 92
 ⟨Initialize fourth column of viewing matrix⟩, 92, 92
 ⟨Initialize Global Variables⟩, 2, 2, 3
 ⟨Initialize interior node and continue recursion⟩, 290, 290
 ⟨Initialize i-th MIPMap level from $i - 1$ st level⟩, 630, 631
 ⟨Initialize leaf node if termination criteria met⟩, 290, 290
 ⟨Initialize levels of MIPMap from image⟩, 625, 630
 ⟨Initialize local variables for selected state⟩, 1040, 1040
 ⟨Initialize material model at sampled surface interaction⟩, 905, 905
 ⟨Initialize mortonPrims[i] for i-th primitive⟩, 271, 272
 ⟨Initialize most detailed level of MIPMap⟩, 630, 630
 ⟨Initialize next ray segment or terminate transmittance computation⟩, 687, 687
 ⟨Initialize pixelBounds and pixels array for SPPM⟩, 973, 973
 ⟨Initialize primitiveInfo array for primitives⟩, 257, 257
 ⟨Initialize primNums for kd-tree construction⟩, 289, 289
 ⟨Initialize prims and areaLights for animated shape⟩, 1124, 1126
 ⟨Initialize prims and areaLights for static shape⟩, 1124, 1124
 ⟨Initialize ProjectionLight projection matrix⟩, 726, 726
 ⟨Initialize sampling PDFs for infinite area light⟩, 739, 847
 ⟨Initialize shading geometry from true geometry⟩, 117, 118
 ⟨Initialize shading partial derivative values⟩, 119, 119
 ⟨Initialize si->bsdf with weighted mixture of BxDFs⟩, 582, 583
 ⟨Initialize sine and cosine iterates⟩, 829, 829
 ⟨Initialize specular component of plastic material⟩, 581, 581
 ⟨Initialize SurfaceInteraction from parametric information⟩, 134, 140, 144, 148
 ⟨Initialize SurfaceInteraction::mediumInterface after Shape intersection⟩, 251, 685
 ⟨Initialize Triangle shading geometry⟩, 165, 166
 ⟨Initialize vertex with surface intersection information⟩, 1007, 1008
 ⟨Integrator Declarations⟩, 25
 ⟨Integrator Interface⟩, 25, 25
 ⟨Integrator Utility Functions⟩, 854, 856, 858, 974
 ⟨Interaction Declarations⟩, 115, 688
 ⟨Interaction Public Data⟩, 115, 115, 115, 116
 ⟨Interaction Public Methods⟩, 115, 115, 116, 232, 687, 688
 ⟨Interpolate (u, v) parametric coordinates and hit point⟩, 157, 164
 ⟨Interpolate camera matrix and check if ω is forward-facing⟩, 950, 950
 ⟨Interpolate rotation at dt⟩, 106, 107
 ⟨Interpolate scale at dt⟩, 106, 107
 ⟨Interpolate translation at dt⟩, 106, 106
 ⟨Interpret the camera subpath as a complete path⟩, 1009, 1009
 ⟨Intersect BSSRDF sampling ray against the scene geometry⟩, 908, 910
 ⟨Intersect ray with curve segment⟩, 174, 176
 ⟨Intersect ray with primitives in leaf BVH node⟩, 283, 283
 ⟨Intersect ray with scene and store intersection in iSect⟩, 877, 877, 900
 ⟨Intersect sample ray with area light geometry⟩, 837, 838
 ⟨Interval Definitions⟩, 112, 113
 ⟨Interval Public Methods⟩, 112, 112, 112
 ⟨Invert definite integral over spline segment and return solution⟩, 823, 824, 825
 ⟨KdAccelNode Methods⟩, 286, 288, 288
 ⟨KdTreeAccel Declarations⟩, 285, 298
 ⟨KdTreeAccel Local Declarations⟩, 286, 291, 292
 ⟨KdTreeAccel Method Definitions⟩, 285, 287, 290, 297
 ⟨KdTreeAccel Private Data⟩, 285, 286, 287, 289
 ⟨KdTreeAccel Public Methods⟩, 285, 302
 ⟨Keep a running sum and build a cumulative distribution function⟩, 937, 938
 ⟨LambertianReflection Private Data⟩, 532, 532
 ⟨LambertianReflection Public Methods⟩, 532, 532, 533
 ⟨LanczosSincFilter Public Methods⟩, 481, 483, 483
 ⟨Launch worker threads if needed⟩, 1088, 1088, 1093
 ⟨Light Declarations⟩, 714, 717, 734
 ⟨Light Interface⟩, 714, 715, 716, 717, 836, 955
 ⟨Light Method Definitions⟩, 718, 718, 741
 ⟨Light Protected Data⟩, 714, 715
 ⟨Light Public Data⟩, 714, 715
 ⟨LightFlags Declarations⟩, 715, 715
 ⟨LightStrategy Declarations⟩, 852
 ⟨Look up connection vertices and their predecessors⟩, 1018, 1018
 ⟨Look up x_i and function values of spline segment i ⟩, 823, 823, 937
 ⟨Loop over filter support and add sample to pixel arrays⟩, 490, 491
 ⟨Loop over pixels in tile to render them⟩, 28, 30
 ⟨Loop over wavelength sample segments and add contributions⟩, 321, 322
 ⟨LoopSubdiv Function Definitions⟩, 184, 196, 198
 ⟨LoopSubdiv Inline Functions⟩, 191, 196, 203
 ⟨LoopSubdiv Local Structures⟩, 185, 186, 188
 ⟨LoopSubdiv Macros⟩, 186
 ⟨Low Discrepancy Data Definitions⟩, 449, 452
 ⟨Low Discrepancy Declarations⟩, 449, 464, 466, 470
 ⟨Low Discrepancy Function Definitions⟩, 444, 449, 450
 ⟨Low Discrepancy Inline Functions⟩, 446, 446, 448, 458, 459, 461, 463, 471
 ⟨Low Discrepancy Static Functions⟩, 447, 450
 ⟨Main program⟩, 21
 ⟨Map ray (p, ω) onto the raster grid⟩, 950, 950, 953
 ⟨Map sampled Beckmann angles to normal direction wh⟩, 808, 809
 ⟨Map u to a spline interval by inverting F⟩, 823, 823
 ⟨Map uniform random numbers to $[-1, 1]^2$ ⟩, 778, 779
 ⟨MarbleTexture Declarations⟩, 663
 ⟨MarbleTexture Private Data⟩, 663, 664

- ⟨MarbleTexture Public Methods⟩, 663, 664, 664
 ⟨Mark connection vertices as non-degenerate⟩, 1018, 1019
 ⟨Material Declarations⟩, 577
 ⟨Material Interface⟩, 577, 577
 ⟨Material Method Definitions⟩, 589
 ⟨Matrix4x4 Declarations⟩, 1081
 ⟨Matrix4x4 Method Definitions⟩, 1080, 1081
 ⟨Matrix4x4 Public Methods⟩, 1081, 1081, 1081
 ⟨MatteMaterial Declarations⟩, 578
 ⟨MatteMaterial Method Definitions⟩, 579
 ⟨MatteMaterial Private Data⟩, 578, 578
 ⟨MatteMaterial Public Methods⟩, 578, 578
 ⟨MaxMinDistSampler Declarations⟩, 465
 ⟨MaxMinDistSampler Method Definitions⟩, 467
 ⟨MaxMinDistSampler Private Data⟩, 465, 466
 ⟨MaxMinDistSampler Public Methods⟩, 465, 466
 ⟨Media Declarations⟩, 681, 702
 ⟨Media Inline Functions⟩, 681
 ⟨Medium Declarations⟩, 684
 ⟨Medium Interface⟩, 684, 684, 891
 ⟨MediumInteraction Public Data⟩, 688, 688
 ⟨MediumInteraction Public Methods⟩, 688, 688, 893
 ⟨MediumInterface Declarations⟩, 684
 ⟨MediumInterface Public Methods⟩, 684, 685, 685
 ⟨Memory Allocation Functions⟩, 1072
 ⟨Memory Declarations⟩, 576, 1072, 1074, 1076
 ⟨MemoryArena Private Data⟩, 1074, 1074, 1075
 ⟨MemoryArena Public Methods⟩, 1074, 1074, 1074, 1076
 ⟨Merge image tile into Film⟩, 28, 32
 ⟨Merge pixel into Film::pixels⟩, 493, 493
 ⟨MicrofacetDistribution Declarations⟩, 537, 538, 540
 ⟨MicrofacetDistribution Method Definitions⟩, 539, 540, 543, 808, 811
 ⟨MicrofacetDistribution Protected Data⟩, 537, 808
 ⟨MicrofacetDistribution Protected Methods⟩, 537, 808
 ⟨MicrofacetDistribution Public Methods⟩, 537, 538, 542, 544, 807
 ⟨MicrofacetReflection Private Data⟩, 547, 547
 ⟨MicrofacetReflection Public Methods⟩, 547, 547
 ⟨MicrofacetTransmission Private Data⟩, 548, 548
 ⟨MicrofacetTransmission Public Methods⟩, 548, 548, 548
 ⟨MIPMap Declarations⟩, 625
 ⟨MIPMap Helper Declarations⟩, 626, 627
 ⟨MIPMap Method Definitions⟩, 625, 631, 632, 634, 635, 637
 ⟨MIPMap Private Data⟩, 625, 625, 630, 639
 ⟨MIPMap Public Methods⟩, 625, 630
 ⟨MitchellFilter Declarations⟩, 479
 ⟨MitchellFilter Method Definitions⟩, 480
 ⟨MitchellFilter Public Methods⟩, 479, 479, 481
 ⟨MixMaterial Declarations⟩, 582
 ⟨MixMaterial Method Definitions⟩, 582
 ⟨MixMaterial Private Data⟩, 582, 582
 ⟨MixMaterial Public Methods⟩, 582, 582
 ⟨MixTexture Declarations⟩, 616
 ⟨MixTexture Public Methods⟩, 616, 616, 617
 ⟨MLT Declarations⟩, 1035
 ⟨MLT Method Definitions⟩, 1036, 1038
 ⟨MLTIntegrator Private Data⟩, 1035, 1037, 1038, 1039, 1040
 ⟨MLTSampler Constants⟩, 1036
 ⟨MLTSampler Declarations⟩, 1029
 ⟨MLTSampler Method Definitions⟩, 1030, 1031, 1032, 1034, 1035
 ⟨MLTSampler Private Data⟩, 1029, 1030, 1030, 1031, 1035
 ⟨MLTSampler Private Declarations⟩, 1029, 1030
 ⟨MLTSampler Public Methods⟩, 1029, 1030, 1035
 ⟨Modify ray for depth of field⟩, 364, 367, 374
 ⟨Normal Declarations⟩, 71, 71
 ⟨Normal3 Public Methods⟩, 71, 72
 ⟨Normalize filter weights for texel resampling⟩, 628, 629
 ⟨Normalize Le values based on maximum blackbody radiance⟩, 711, 711
 ⟨Normalize pixel with weight sum⟩, 495, 495
 ⟨Notify worker threads of work to be done⟩, 1088, 1090
 ⟨Offset ray origin to edge of error bounds and compute tMax⟩, 95, 233
 ⟨OrenNayar Private Data⟩, 536
 ⟨OrenNayar Public Methods⟩, 536
 ⟨OrthographicCamera Declarations⟩, 361
 ⟨OrthographicCamera Definitions⟩, 364, 364
 ⟨OrthographicCamera Private Data⟩, 361, 363
 ⟨OrthographicCamera Public Methods⟩, 361, 361
 ⟨Override surface normal in intersect for triangle⟩, 165, 165
 ⟨Parallel Declarations⟩, 1086, 1089, 1093
 ⟨Parallel Definitions⟩, 1088, 1093
 ⟨Parallel Local Definitions⟩, 1088, 1089, 1090, 1092
 ⟨ParallelForLoop Private Data⟩, 1090, 1090, 1090
 ⟨ParallelForLoop Private Methods⟩, 1090, 1090
 ⟨ParallelForLoop Public Methods⟩, 1090, 1090
 ⟨ParamSet Declarations⟩, 1105, 1106
 ⟨ParamSet Methods⟩, 1107, 1108
 ⟨ParamSet Private Data⟩, 1105, 1105
 ⟨ParamSet Public Methods⟩, 1105, 1107, 1107, 1108, 1109
 ⟨ParamsetItem Data⟩, 1106, 1106
 ⟨ParamsetItem Methods⟩, 1106
 ⟨Parse scene from input files⟩, 21, 22
 ⟨Parse scene from standard input⟩, 21, 22
 ⟨Partition primitives into equally sized subsets⟩, 262, 265
 ⟨Partition primitives into two sets and build children⟩, 259, 261
 ⟨Partition primitives through node's midpoint⟩, 262
 ⟨Partition primitives using approximate SAH⟩, 265
 ⟨PathIntegrator Declarations⟩, 875
 ⟨PathIntegrator Method Definitions⟩, 876
 ⟨PathIntegrator Private Data⟩, 875, 876
 ⟨PathIntegrator Public Methods⟩, 875, 876
 ⟨Perform a Newton step⟩, 825, 828
 ⟨Perform bump mapping with bumpMap, if present⟩, 579, 579, 581, 584, 701
 ⟨Perform connection and write contribution to L⟩, 1009, 1009
 ⟨Perform nIterations of SPPM integration⟩, 973, 975
 ⟨Perform one pass of radix sort, sorting bitsPerPass bits⟩, 274, 274
 ⟨Perform projective divide for perspective projection⟩, 365, 366
 ⟨Perform ratio tracking to estimate the transmittance value⟩, 898, 898
 ⟨Perform ray-triangle intersection test⟩, 157, 158
 ⟨Perform triangle edge and determinant tests⟩, 158, 162

- (Perform trilinear interpolation at appropriate MIPMap level), 632, 633
 (Perlin Noise Data), 652
 (Permute components of triangle vertices and ray direction), 158, 159
 (Permute LHS samples in each dimension), 440, 440
 (PerspectiveCamera Declarations), 365
 (PerspectiveCamera Method Definitions), 365, 367, 950, 953, 954
 (PerspectiveCamera Private Data), 365, 367, 951
 (PerspectiveCamera Public Methods), 365, 367
 (PhaseFunction Interface), 681, 681, 898
 (Pick a side and declare bisection variables), 828, 829
 (PixelSampler Protected Data), 427, 427, 428
 (PlanarMapping2D Public Methods), 613, 613
 (PlasticMaterial Declarations), 580
 (PlasticMaterial Method Definitions), 581
 (PlasticMaterial Private Data), 580, 580
 (PlasticMaterial Public Methods), 580, 580
 (Point Declarations), 67, 68
 (Point sample Checkerboard2DTexture), 643, 643, 645
 (Point2 Public Data), 67, 68
 (Point2 Public Methods), 67, 68
 (Point3 Public Data), 68, 68
 (Point3 Public Methods), 68, 68, 69, 70
 (PointLight Declarations), 719
 (PointLight Method Definitions), 721, 721, 836, 955, 956
 (PointLight Private Data), 719, 720
 (PointLight Public Methods), 719, 720
 (Populate arguments and compute the importance value), 954, 954
 (Populate mi with medium interaction information and return), 897, 897
 (Possibly add emitted light at intersection), 877, 877
 (Possibly create area light for shape), 1124, 1125
 (Possibly create visible point and end camera path), 978, 979
 (Possibly terminate photon path with Russian roulette), 987, 989
 (Possibly terminate the path with Russian roulette), 877, 879, 900
 (Potentially flip ϕ and return the result), 828, 830
 (Potentially swap indices of refraction), 519, 519
 (Precompute filter weight table), 484, 487
 (Precompute information for dipole integrand), 928, 928
 (Precompute values for Monte Carlo sampling of GridDensityMedium), 690, 896
 (Precompute x and y filter table offsets), 491, 492
 (Prepare for next level of subdivision), 194, 203
 (Prepare tileSampler for pPixel), 976, 977
 (Prepare to traverse kd-tree for ray), 297, 298
 (PrimarySample Public Data), 1030, 1032, 1034
 (PrimarySample Public Methods), 1030, 1034
 (Primitive Declarations), 248
 (Primitive Interface), 248, 249, 249, 250
 (Process kd-tree interior node), 299, 299
 (Process scene description), 21, 21
 (Process SPPM camera ray intersection), 977, 978
 (ProfilePhase Private Data), 1099, 1099
 (ProfilePhase Public Methods), 1099, 1099
 (ProgressReporter Public Methods), 1069, 1069, 1070
 (Project curve control points to plane perpendicular to ray), 173, 173
 (Project point onto projection plane and compute light), 727, 728
 (ProjectionLight Declarations), 724
 (ProjectionLight Method Definitions), 726, 727, 728
 (ProjectionLight Private Data), 724, 726, 727
 (ProjectiveCamera Protected Data), 358, 360, 360, 374
 (ProjectiveCamera Public Methods), 358, 360
 (Push vertices to limit surface), 193, 203
 (Put far BVH node on nodesToVisit stack, advance to near node), 283, 284
 (Put vert one-ring in pRing), 196, 196, 198
 (Quaternion Inline Functions), 100, 100
 (Quaternion Method Definitions), 103
 (Quaternion Public Data), 100
 (Quaternion Public Methods), 99, 100, 101
 (Radix sort primitive Morton indices), 271, 273
 (Random Number Declarations), 417
 (Randomly choose a single light to sample, light), 856, 856
 (Randomly choose one of several intersections during BSSRDF sampling), 908, 911
 (Randomly shuffle 1D sample points), 463, 464
 (Ray Declarations), 73, 75
 (Ray Public Data), 73, 73, 73, 74
 (Ray Public Methods), 73, 74, 74
 (RayDifferential Public Data), 75, 75
 (RayDifferential Public Methods), 75, 75, 75
 (Re-scale u for continuous spline sampling step), 823, 824
 (RealisticCamera Declarations), 378
 (RealisticCamera Method Definitions), 379, 382, 383, 387, 388, 391, 393, 394
 (RealisticCamera Private Data), 378, 379, 381
 (RealisticCamera Private Declarations), 378, 381
 (RealisticCamera Private Methods), 378, 381, 382, 385
 (Record medium interaction in path and compute forward density), 1006, 1007
 (Recursively initialize children nodes), 290, 297
 (Refine cylinder intersection point), 145, 225
 (Refine disk intersection point), 148, 225
 (Refine sphere intersection point), 137, 225
 (Refine subdivision mesh into triangles), 184, 193
 (Reflection Declarations), 521
 (Reject disk intersections for rays parallel to the disk's plane), 148, 149
 (Remap BxDF sample u to $[0, 1]^2$), 832, 833
 (Remap Sobol' dimensions used for pixel samples), 471, 472
 (Render image tiles in parallel), 26, 27
 (Render section of image corresponding to tile), 27, 28
 (RenderOptions Public Data), 1114, 1115, 1116, 1117, 1123, 1127, 1128
 (RenderOptions Public Methods), 1114, 1130
 (Report thread statistics at worker thread exit), 1092, 1093
 (Reproject pObj to cylinder surface and compute pObjError), 839, 839
 (Reproject pObj to sphere surface and compute pObjError), 840, 840, 844

- (Request samples for sampling all lights)*, 852, 853
(Resample image in s direction), 627, 627
(Resample image to power-of-two resolution), 625, 627
(Reset array offsets for next pixel sample), 425, 426
(Reset X_i if a large step took place in the meantime), 1032, 1032
(Reset VisiblePoint in pixel), 989, 990
(Return BSSRDF profile density for channel ch), 914, 914
(Return combined probability from all BSSRDF sampling strategies), 912, 913
(Return contribution of specular reflection), 37, 38
(Return emitted radiance for infinite light sources), 999, 1000
(Return entire element bounds if no rays made it through the lens system), 391, 393
(Return false if x is out of bounds), 562, 562
(Return importance for point on image plane), 950, 953
(Return insideDot result if point is inside dot), 654, 655
(Return Interaction for sampled point on sphere), 841, 844
(Return MIPMap from texture cache if present), 620, 620
(Return motion bounds accounting for animated rotation), 108, 108
(Return probability per unit area at vertex next), 1001, 1001
(Return radiance value for infinite light direction), 849, 850
(Return raster position if requested), 950, 950
(Return sample point rotated by angle of pFilm with +x axis), 393, 394
(Return solid angle density for infinite light sources), 1003, 1021
(Return solid angle density for non-infinite light sources), 1003, 1003
(Return solid angle density if next is an infinite area light), 1000, 1020
(Return the sample position and function value), 825, 828
(Return uniform PDF if point is inside sphere), 844, 844
(Return weighting factor for scattering from homogeneous medium), 893, 894
(Return weighting for RealisticCamera ray), 394, 783
- (Return x $\in [0, 1]$ corresponding to sample)*, 759, 760
(Return zero importance for out of bounds points), 950, 951
(RGBSpectrum Public Methods), 332, 332, 332, 333
(RNG Public Methods), 1065, 1066
(Rotate dpdvPlane to give cylindrical appearance), 181, 181
(Round offset point po away from p), 231, 232
(Round up nBytes to minimum machine alignment), 1074, 1075
(Run a chunk of loop iterations for loop), 1091, 1091, 1093
(Run delta-tracking iterations to sample a medium interaction), 896, 897
(Run iterations immediately if not using threads or if count is small), 1088, 1088
(Run loop indices in [indexStart, indexEnd]), 1091, 1092
(Run nChains Markov chains in parallel), 1038, 1039
(Run the Markov chain for nChainMutations steps), 1040, 1041
(Sample a channel and distance along the ray), 893, 894
(Sample a collection of points on the rear lens to find exit pupil), 391, 392
(Sample a cosine-weighted outgoing direction w for area light), 957, 957
(Sample a point on the area light's Shape, pShape), 957, 957
(Sample a point on the camera and connect it to the light subpath), 1009, 1010
(Sample BSDF at current vertex and compute reverse probability), 1007, 1008
(Sample BSDF fr and direction wi for reflected photon), 987, 988
(Sample BSDF to get new path direction), 877, 878
(Sample BSDF with multiple importance sampling), 858, 860
(Sample BSSRDF profile in polar coordinates), 908, 909
(Sample chosen BxDF), 832, 833
(Sample direction and compute reverse density at preceding vertex), 1006, 1007
(Sample full distribution of normals for Beckmann distribution), 808, 808
(Sample illumination from lights to find path contribution), 877, 878
- (Sample initial ray for camera subpath)*, 1003, 1004
(Sample initial ray for light subpath), 1004, 1005
(Sample light source with multiple importance sampling), 858, 858
(Sample microfacet orientation ω_h and reflected direction ω_i), 811, 812, 814
(Sample new photon ray direction), 986, 987
(Sample point on lens), 374, 374
(Sample real point source depth z_r), 928, 929
(Sample scattered direction for medium interactions), 860, 900
(Sample scattered direction for surface interactions), 860, 860
(Sample sphere uniformly inside subtended cone), 841, 841
(Sample the participating medium, if present), 900, 901
(Sample the standard normal distribution $N(0, 1)$), 1033, 1034
(Sample uniformly on sphere if p is inside it), 841, 841
(Sample zenith angle component for FourierBSDF), 819, 819
(SampledSpectrum Private Data), 319, 324, 329, 330
(SampledSpectrum Public Methods), 319, 319, 319, 324, 325, 328, 331
(Sampler Declarations), 421, 427, 429
(Sampler Interface), 421, 422, 422, 423, 424
(Sampler Method Definitions), 422, 423, 425, 426, 427, 428, 430, 431, 432
(Sampler Private Data), 421, 426
(Sampler Protected Data), 421, 425, 426
(Sampler Public Data), 421, 422
(SamplerIntegrator Declarations), 25
(SamplerIntegrator Method Definitions), 26, 37
(SamplerIntegrator Private Data), 25, 25
(SamplerIntegrator Protected Data), 25, 26
(SamplerIntegrator Public Methods), 25, 26, 26, 31
(Sampling Declarations), 758, 785
(Sampling Function Definitions), 437, 437, 440, 762, 775, 776, 777, 778, 781, 782, 785
(Sampling Inline Functions), 438, 780, 799
(Save final image after rendering), 26, 32
(Scale canonical perspective view to specified field of view), 365, 367

- `<Scale hitWidth based on ribbon orientation>`, 179, 179
`<Scale pixel value by scale>`, 495, 496
`<ScaledBxDF Public Methods>`, 515, 515
`<ScaleTexture Declarations>`, 615
`<ScaleTexture Private Data>`, 615, 616
`<ScaleTexture Public Methods>`, 615, 616, 616
`<Scan over ellipse bound and compute quadratic equation>`, 637, 639
`<Scene Constructor Implementation>`, 23, 24, 24
`<Scene Declarations>`, 23
`<Scene Method Definitions>`, 24, 24, 687
`<Scene Private Data>`, 23, 23, 24
`<Scene Public Data>`, 23, 23
`<Scene Public Methods>`, 23, 23, 24
`<ScopedAssignment Public Methods>`, 1017, 1018
`<SDEdge Comparison Function>`, 188, 188
`<SDEdge Constructor>`, 188, 188
`<SDFace Methods>`, 186, 191, 192, 200
`<SDVertex Constructor>`, 185, 185
`<Search for the interval idx containing x>`, 562, 562
`<See if hit point is inside disk radii and φmax>`, 148, 149
`<Select initial state from the set of bootstrap samples>`, 1040, 1040
`<Select intersection t based on ray direction and element curvature>`, 383, 384
`<SeparableBSSRDF Interface>`, 693, 695, 909
`<SeparableBSSRDF Private Data>`, 693, 693
`<SeparableBSSRDF Public Methods>`, 693, 693, 694, 695
`<SeparableBSSRDFAdapter Public Methods>`, 906, 906, 906
`<Set BSSRDF value Sr[ch] using tensor spline interpolation>`, 699, 699
`<Set face to vertex pointers>`, 184, 187
`<Set in and out vector pointers for radix sort pass>`, 274, 274
`<Set initial guess for t by importance sampling a linear interpolant>`, 825, 826
`<Set neighbor pointers in faces>`, 184, 188
`<Set ray origin and direction for infinite light ray>`, 958, 959
`<Set spatial density of path[0] for infinite area light>`, 1021, 1021
`<Set spatial density of path[1] for infinite area light>`, 1021, 1021
`<Shape Declarations>`, 123
`<Shape Interface>`, 123, 125, 129, 130, 131, 837
`<Shape Method Definitions>`, 124, 125, 837
`<Shape Public Data>`, 123, 124
`<Shift sEval du in the u direction>`, 590, 590
`<Sinc Filter Declarations>`, 481
`<Sinc Filter Method Definitions>`, 483
`<Sleep until there are more tasks to run>`, 1092, 1092
`<Sobol Matrix Declarations>`, 471
`<SobolSampler Declarations>`, 468
`<SobolSampler Method Definitions>`, 470, 471
`<SobolSampler Private Data>`, 468, 470
`<SobolSampler Public Methods>`, 468, 470
`<Solve quadratic equation for t values>`, 134, 136, 141, 144
`<Sort edges for axis>`, 293, 293
`<Sort samples if unordered, use sorted for returned spectrum>`, 319, 320, 333
`<Spectral Data Declarations>`, 323, 325, 329
`<Spectrum Declarations>`, 316, 319, 332
`<Spectrum Inline Functions>`, 317
`<Spectrum Method Definitions>`, 321, 330, 331, 333, 710, 711
`<Spectrum Utility Declarations>`, 319, 327, 328, 330
`<SpecularReflection Private Data>`, 524, 524
`<SpecularReflection Public Methods>`, 524, 524, 816
`<SpecularTransmission Private Data>`, 528, 529
`<SpecularTransmission Public Methods>`, 528, 528, 529, 816
`<Sphere Declarations>`, 133
`<Sphere Method Definitions>`, 134, 134, 141, 142, 840, 841, 844, 1131
`<Sphere Private Data>`, 133, 133
`<Sphere Public Methods>`, 133, 133
`<Splat both current and proposed samples to film>`, 1041, 1041
`<Spline Interpolation Declarations>`, 939
`<Spline Interpolation Definitions>`, 562, 823, 824, 937
`<Split curve segment into sub-segments and test for intersection>`, 174, 175
`<Split tInterval and check both resulting intervals>`, 113, 114
`<SpotLight Declarations>`, 721
`<SpotLight Method Definitions>`, 723, 723, 724, 956
`<SpotLight Private Data>`, 721, 723
`<SPPM Declarations>`, 973
`<SPPM Local Definitions>`, 974, 980, 982
`<SPPM Method Definitions>`, 973
`<SPPMIntegrator Private Data>`, 973, 973, 975, 977, 983
`<SPPMPixel Public Data>`, 974, 974, 978, 979, 987, 990
`<Start recursive construction of kd-tree>`, 289, 289
`<Statistics Declarations>`, 1098, 1099
`<Statistics Definitions>`, 1097, 1097, 1100
`<Statistics Local Variables>`, 1097, 1100
`<Statistics Macros>`, 1095
`<StatRegisterer Private Data>`, 1096
`<StatRegisterer Public Methods>`, 1096
`<StatsAccumulator Private Data>`, 1096
`<StatsAccumulator Public Methods>`, 1096
`<Stop the iteration if converged>`, 825, 827
`<Store final image computed with MLT>`, 1038, 1042
`<Store high-precision reference value in EFloat>`, 218, 219
`<Store primitive ids for leaf node>`, 287, 287
`<Store sorted values in output array>`, 274, 275
`<StratifiedSampler Declarations>`, 434
`<StratifiedSampler Method Definitions>`, 434
`<StratifiedSampler Private Data>`, 434, 434
`<StratifiedSampler Public Methods>`, 434, 434
`<SubsurfaceMaterial Declarations>`, 700
`<SubsurfaceMaterial Method Definitions>`, 701
`<SubsurfaceMaterial Private Data>`, 700, 701
`<SurfaceInteraction Declarations>`, 116
`<SurfaceInteraction Method Definitions>`, 117, 119, 578, 601, 734
`<SurfaceInteraction Public Data>`, 116, 116, 118, 249, 250, 600
`<TabulatedBSSRDF Private Data>`, 696, 697
`<TabulatedBSSRDF Public Methods>`, 696, 697
`<Temporarily update vertex properties for current strategy>`, 1016, 1018
`<Terminate path if ray escaped or maxDepth was reached>`, 877, 877
`<Test cylinder intersection against clipping parameters>`, 144, 145
`<Test disk φ value against φmax>`, 149, 149
`<Test intersection against alpha texture, if present>`, 157, 165

- ⟨Test intersection point against curve width⟩, 176, 179
 ⟨Test intersection point against element aperture⟩, 383, 384
 ⟨Test ray against segment endpoint boundaries⟩, 176, 176
 ⟨Test sample point against tangent perpendicular at curve start⟩, 176, 177
 ⟨Test sphere intersection against clipping parameters⟩, 134, 137, 141
 ⟨Texture Declarations⟩, 609, 610, 611, 612, 613, 614, 629
 ⟨Texture Inline Functions⟩, 658
 ⟨Texture Interface⟩, 614, 615
 ⟨Texture Method Definitions⟩, 610, 610, 611, 612, 613, 614, 649, 652, 653, 656, 660
 ⟨TextureMapping2D Interface⟩, 609, 610
 ⟨TextureMapping3D Interface⟩, 614, 614
 ⟨TextureParams Declarations⟩, 1120
 ⟨TextureParams Method Definitions⟩, 1121
 ⟨TextureParams Private Data⟩, 1120, 1120
 ⟨TextureParams Public Methods⟩, 1120, 1121
 ⟨Trace a ray and sample the medium, if any⟩, 1006, 1006
 ⟨Trace photons and accumulate contributions⟩, 975, 983
 ⟨Trace ray from pFilm through lens system⟩, 394, 394
 ⟨Trace rays for specular reflection and refraction⟩, 34, 37
 ⟨Trace the camera and light subpaths⟩, 994, 994
 ⟨Transform BSSRDF value into world space units⟩, 699, 700
 ⟨Transform Declarations⟩, 83
 ⟨Transform Inline Functions⟩, 93, 93, 95, 228
 ⟨Transform instance’s intersection data to world space⟩, 253, 253
 ⟨Transform mesh vertices to world space⟩, 155, 155
 ⟨Transform Method Definitions⟩, 87, 87, 89, 91, 92, 95, 96, 120, 363, 365
 ⟨Transform p and pError in SurfaceInteraction⟩, 120, 229
 ⟨Transform Private Data⟩, 83, 84
 ⟨Transform Public Methods⟩, 83, 84, 84, 85, 88, 229
 ⟨Transform Ray to object space⟩, 134, 134, 141, 144, 148, 173
 ⟨Transform rCamera from camera to lens system space⟩, 382, 383
 ⟨Transform rLens from lens system space back to camera space⟩, 382, 385
 ⟨Transform step function integral into CDF⟩, 758, 759
 ⟨Transform triangle vertices to ray coordinate space⟩, 158, 158
 ⟨TransformCache Private Data⟩, 1125
 ⟨TransformedPrimitive Declarations⟩, 252
 ⟨TransformedPrimitive Method Definitions⟩, 253
 ⟨TransformedPrimitive Private Data⟩, 252, 252
 ⟨TransformedPrimitive Public Methods⟩, 252, 252, 254
 ⟨TransformSet Public Methods⟩, 1112, 1112, 1112, 1112, 1114
 ⟨Translate vertices based on ray origin⟩, 158, 158
 ⟨TransportMode Declarations⟩, 960
 ⟨Traverse kd-tree nodes in order for ray⟩, 297, 299
 ⟨Triangle Declarations⟩, 154, 156
 ⟨Triangle Method Definitions⟩, 155, 156, 157, 167, 839
 ⟨Triangle Private Data⟩, 156, 156
 ⟨Triangle Private Methods⟩, 156, 164
 ⟨Triangle Public Methods⟩, 156, 156
 ⟨TriangleFilter Declarations⟩, 477
 ⟨TriangleFilter Method Definitions⟩, 477
 ⟨TriangleMesh Data⟩, 154, 155
 ⟨Trilinearly interpolate density values to compute local density⟩, 691, 691
 ⟨TrowbridgeReitzDistribution Private Data⟩, 540, 540
 ⟨TrowbridgeReitzDistribution Public Methods⟩, 540, 540
 ⟨Try to cull curve segment versus ray⟩, 174, 174
 ⟨Try to get memory block from availableBlocks⟩, 1075, 1075
 ⟨Uniformly sample a lens interaction lensIntr⟩, 954, 954
 ⟨Update best split if this is lowest cost so far⟩, 295, 295
 ⟨Update bisection bounds using updated t⟩, 825, 828
 ⟨Update BSSRDF transmission term to account for adjoint light transport⟩, 906, 961
 ⟨Update child vertex pointer to new even vertex⟩, 202, 202
 ⟨Update child vertex pointer to new odd vertex⟩, 202, 202
 ⟨Update children f pointers for neighbor children⟩, 201, 201
 ⟨Update children f pointers for siblings⟩, 201, 201
 ⟨Update even vertex face pointers⟩, 201, 201
 ⟨Update f and v for next level of subdivision⟩, 193, 194
 ⟨Update face neighbor pointers⟩, 201, 201
 ⟨Update face vertex pointers⟩, 201, 202
 ⟨Update interval for i-th bounding box slab⟩, 127, 128
 ⟨Update loop to reflect completion of iterations⟩, 1091, 1092
 ⟨Update loop to reflect iterations this thread will run⟩, 1091, 1091
 ⟨Update neighbor pointer for edgeNum⟩, 188, 189
 ⟨Update new mesh topology⟩, 194, 201
 ⟨Update parametric interval from slab intersection t values⟩, 128, 128
 ⟨Update pixel Φ and M for nearby photon⟩, 986, 987
 ⟨Update pixel photon count, search radius, and τ from photons⟩, 989, 989
 ⟨Update pixel values from this pass’s photons⟩, 975, 989
 ⟨Update pixel values with filtered sample contribution⟩, 491, 492
 ⟨Update ray for effect of lens⟩, 374, 375
 ⟨Update ray from film accounting for interaction with element⟩, 382, 383
 ⟨Update ray path for element interface interaction⟩, 383, 385
 ⟨Update reverse density of vertex $p_{t-1}
 ⟨Update reverse density of vertex $p_{t-2}
 ⟨Update sampled vertex for $s = 1$ or $t = 1$ strategy⟩, 1018, 1018
 ⟨Update scale to account for adjoint light transport⟩, 558, 961
 ⟨Update tFar to ensure robust ray–bounds intersection⟩, 128, 221
 ⟨Update tHit for quadric intersection⟩, 134, 140, 144, 148
 ⟨Update transmittance for current ray segment⟩, 718, 719
 ⟨Update vertex positions and create new edge vertices⟩, 194, 194
 ⟨Update vertex positions for even vertices⟩, 194, 195
 ⟨Use a single sample for illumination from light⟩, 855, 855
 ⟨Use Newton’s method to refine zero⟩, 113, 114
 ⟨UVTexture Declarations⟩, 641
 ⟨UVTexture Public Methods⟩, 641, 641$$

(*Vector Declarations*), 59, 59, 60
(*Vector2 Public Data*), 59, 59
(*Vector3 Public Data*), 59, 59
(*Vector3 Public Methods*), 59, 60, 61, 62,
 63, 65
(*Vertex Public Data*), 996, 996, 996, 998,
 1000
(*Vertex Public Methods*), 996, 997, 997,
 998, 999, 1000, 1001, 1002, 1003
(*VisibilityTester Public Methods*), 717,
 717, 718
(*VolPathIntegrator Method Definitions*),
 900
(*WhittedIntegrator Declarations*), 32
(*WhittedIntegrator Method Definitions*),
 33
(*WhittedIntegrator Private Data*), 32, 33
(*WhittedIntegrator Public Methods*), 32,
 33
(*WindyTexture Declarations*), 662
(*WindyTexture Public Methods*), 662, 662,
 663
(*Write RGB image*), 494, 496
(*ZeroTwoSequenceSampler Declarations*),
 462
(*ZeroTwoSequenceSampler Method
Definitions*), 462, 463
(*ZeroTwoSequenceSampler Public
Methods*), 462, 462

This page intentionally left blank

INDEX OF CLASSES AND THEIR MEMBERS

Bold numbers indicate the page of a class definition. Class methods and fields are indented.

AAMethod, 643	maxDepth, 993	BoundEdge, 292
ClosedForm, 643	Render(), 994	primNum, 292
None, 643	sampler, 993	t, 292
Aggregate, 255	BeckmannDistribution, 538	type, 292
AllTransformsBits, 1112	alphax, 539	Bounds2, 76
AnimatedTransform, 103	alphay, 539	Diagonal(), 80
actuallyAnimated, 103	D(), 539	Expand(), 80
BoundPointMotion(), 110	Lambda(), 543	Inside(), 79
Decompose(), 104	Sample_wh(), 808	InsideExclusive(), 79
endTime, 103	BilerpTexture, 617	Intersect(), 78
endTransform, 103	Evaluate(), 618	Lerp(), 80
hasRotation, 103	mapping, 617	Overlaps(), 79
Interpolate(), 106	v00, 617	Union(), 78
MotionBounds(), 108	v01, 617	Bounds2f, 76
R, 103	v10, 617	Bounds2i, 76
S, 103	v11, 617	Bounds3, 76
startTime, 103	BlockedArray, 1076	BoundingSphere(), 81
startTransform, 103	Block(), 1078	Corner(), 78
T, 103	BlockSize(), 1078	Diagonal(), 80
APIState, 1110	data, 1078	Expand(), 79
OptionsBlock, 1110	logBlockSize, 1076	Inside(), 79
Uninitialized, 1110	Offset(), 1078	InsideExclusive(), 79
WorldBlock, 1110	operator(), 1079	Intersect(), 78
AreaLight, 734	RoundUp(), 1078	IntersectP(), 127
L(), 734	uBlocks, 1078	Lerp(), 80
AtomicFloat, 1086	uRes, 1078	MaximumExtent(), 80
Add(), 1087	uSize(), 1078	Offset(), 81
bits, 1086	vRes, 1078	operator[], 77
BDPTIntegrator, 992	vSize(), 1078	Overlaps(), 79
camera, 993		pMax, 77

Bounds3 (continued)
 pMin, 77
 SurfaceArea(), 80
 Union(), 78
 Volume(), 80
Bounds3f, 76
Bounds3i, 76
BoxFilter, 477
 Evaluate(), 477
BSDF, 572
 Add(), 573
 bxdfs, 573
 eta, 573
 f(), 575
 LocalToWorld(), 574
 MaxBxDFs, 573
 nBxDFs, 573
 ng, 573
 ns, 573
 NumComponents(), 573
 Pdf(), 834
 rho(), 575
 Sample_f(), 832
 ss, 573
 ts, 573
 WorldToLocal(), 574
BSSRDF, 692
 eta, 692
 po, 692
 S(), 693
 Sample_S(), 904
BSSRDFTable, 697
 EvalProfile(), 700
 nRadiusSamples, 698
 nRhoSamples, 698
 profile, 698
 profileCDF, 914
 radiusSamples, 698
 rhoEff, 698
 rhoSamples, 698
 bounds, 266
 count, 266
BVHAccel, 256
 buildUpperSAH(), 280
 emitLBVH(), 278
 flattenBVHTree(), 282
 HLBVHBuild(), 271
 Intersect(), 282
 IntersectP(), 284
 maxPrimsInNode, 257
 nodes, 281
 primitives, 257
 recursiveBuild(), 259
 splitMethod, 257
BVHBuildNode, 258
 bounds, 258

children, 258
 firstPrimOffset, 258
 InitInterior(), 259
 InitLeaf(), 258
 nPrimitives, 258
 splitAxis, 258
BVHPrimitiveInfo, 257
 bounds, 257
 centroid, 257
 primitiveNumber, 257
BxDF, 513
 f(), 514
 MatchesFlags(), 513
 Pdf(), 807
 rho(), 515
 Sample_f(), 806
 type, 513
BxDFTType, 513
Camera, 356
 CameraToWorld, 356
 film, 356
 GenerateRay(), 357
 GenerateRayDifferential(), 357
 medium, 356
 Pdf_We(), 953
 Sample_Wi(), 954
 shutterClose, 356
 shutterOpen, 356
 We(), 949
CameraSample, 357
 pFilm, 357
 pLens, 357
 time, 357
Checkerboard2DTexture, 642
 aaMethod, 643
 Evaluate(), 643
 mapping, 643
 tex1, 643
 tex2, 643
Checkerboard3DTexture, 647
 Evaluate(), 648
 mapping, 648
 tex1, 648
 tex2, 648
CMaxMinDist, 466
CoefficientSpectrum, 316
 c, 316
 nSamples, 318
Cone, 150
ConstantTexture, 615
 Evaluate(), 615
Curve, 168
 Area(), 181
 common, 170
 Intersect(), 173
 ObjectBound(), 171
 recursiveIntersect(), 174
 uMax, 170
 uMin, 170
CurveCommon, 171
 cpObj, 171
 invSinNormalAngle, 171
 n, 171
 normalAngle, 171
 type, 171
 width, 171
CurveType, 169
Cylinder, 169
Flat, 169
Ribbon, 169
Cylinder, 142
 Area(), 146
 Intersect(), 144
 ObjectBound(), 143
 phiMax, 143
 radius, 143
 Sample(), 839
 zMax, 143
 zMin, 143
CylindricalMapping2D, 612
 cylinder(), 613
Map(), 612
 WorldToTexture, 612
DerivativeTerm, 110
DiffuseAreaLight, 736
 area, 736
 L(), 736
 Lemit, 736
 Pdf_Li(), 845
 Power(), 736
 Sample_Le(), 957
 Sample_Li(), 845
 shape, 736
DirectLightingIntegrator, 851
 Li(), 853
 maxDepth, 852
 nLightSamples, 853
 Preprocess(), 852
 strategy, 852
Disk, 146
 Area(), 150
 height, 147
 innerRadius, 147
 Intersect(), 148
 ObjectBound(), 148
 phiMax, 147
 radius, 147
 Sample(), 838
DistantLight, 731
 L, 731
 Power(), 733
 Preprocess(), 732

Sample_Le(), 958
Sample_Li(), 732
wLight, 731
worldCenter, 732
worldRadius, 732
Distribution1D, 758
cdf, 758
Count(), 758
DiscretePDF(), 760
func, 758
funcInt, 758
SampleContinuous(), 759
SampleDiscrete(), 760
Distribution2D, 785
pConditionalV, 785
Pdf(), 787
pMarginal, 785
SampleContinuous(), 786
DotsTexture, 653
Evaluate(), 654
insideDot, 653
mapping, 653
outsideDot, 653
EdgeType, 291
End, 291
Start, 291
EFloat, 218
err, 218
GetAbsoluteError(), 220
GetRelativeError(), 220
1d, 219
LowerBound(), 220
PreciseValue(), 220
UpperBound(), 220
v, 218
EndpointInteraction, 996
camera, 996
EndpointInteraction(), 997
light, 996
EndTransformBits, 1112
EnvironmentCamera, 376
GenerateRay(), 377
FBmTexture, 660
Evaluate(), 662
mapping, 660
octaves, 660
omega, 660
Film, 484
AddSplat(), 494
croppedPixelBounds, 485
diagonal, 485
filename, 485
filter, 485
filterTable, 487
filterTableWidth, 487
fullResolution, 485
GetFilmTile(), 488
GetPhysicalExtent(), 488
GetPixel(), 493
GetSampleBounds(), 487
MergeFilmTile(), 493
mutex, 493
pixels, 486
scale, 496
SetImage(), 494
WriteImage(), 494
FilmTile, 489
AddSample(), 490
filterRadius, 489
filterTable, 489
filterTableSize, 489
GetPixel(), 492
GetPixelBounds(), 493
invFilterRadius, 489
pixelBounds, 489
pixels, 489
FilmTilePixel, 489
contribSum, 489
filterWeightSum, 489
Filter, 474
Evaluate(), 475
invRadius, 475
radius, 475
Float, 1062
FourierBSDF, 555
bsdfTable, 556
f(), 556
mode, 556
Pdf(), 821
Sample_f(), 819
FourierBSDFTable, 554
a, 555
a0, 818
aOffset, 555
cdf, 819
eta, 554
GetAk(), 555
GetWeightsAndOffset(), 563
m, 555
mMax, 554
mu, 554
nChannels, 554
nMu, 554
Read(), 554
recip, 820
FourierMaterial, 583
bsdfTable, 583
bumpMap, 583
ComputeScatteringFunctions(), 584
Fresnel, 521
Evaluate(), 522
FresnelBlend, 550
distribution, 550
f(), 551
Pdf(), 815
Rd, 550
Rs, 550
Sample_f(), 814
SchlickFresnel(), 551
FresnelConductor, 522
etaI, 522
etaT, 522
Evaluate(), 522
FresnelConductor(), 522
k, 522
FresnelDielectric, 522
etaI, 522
etaT, 522
Evaluate(), 523
FresnelDielectric(), 522
FresnelNoOp, 523
Evaluate(), 523
FresnelSpecular, 531
etaA, 532
etaB, 532
f(), 532
fresnel, 532
mode, 532
R, 532
Sample_f(), 816
T, 532
GaussianFilter, 478
alpha, 478
Evaluate(), 479
expX, 478
expY, 478
GeometricPrimitive, 250
arealight, 250
ComputeScatteringFunctions(), 251
GetAreaLight(), 251
Intersect(), 251
IntersectP(), 251
material, 250
mediumInterface, 250
shape, 250
WorldBound(), 251
GlassMaterial, 584
GlobalSampler, 429
arrayEndDim, 431
arrayStartDim, 431
dimension, 430
Get1D(), 432
Get2D(), 432
GetIndexForSample(), 430
intervalSampleIndex, 430
SampleDimension(), 430
SetSampleNumber(), 431

GlobalSampler (*continued*)
 StartNextSample(), 431
 StartPixel(), 430

GonioPhotometricLight, 728
 I, 730
 mipmap, 730
 plight, 730
 Power(), 731

GraphicsState, 1118
 areaLight, 1123
 areaLightParams, 1123
 CreateMaterial(), 1125
 CreateMediumInterface(), 1125
 currentInsideMedium, 1117
 currentNamedMaterial, 1123
 currentOutsideMedium, 1117
 floatTextures, 1122
 material, 1122
 materialParams, 1122
 namedMaterials, 1123
 spectrumTextures, 1122

GridDensityMedium, 690
 D(), 691
 Density(), 691
 g, 690
 GridDensityMedium(), 690
 invMaxDensity, 896
 nx, 690
 ny, 690
 nz, 690
 Sample(), 896
 sigma_a, 690
 sigma_s, 690
 sigma_t, 896
 Tr(), 898
 WorldToMedium, 690

HaltonSampler, 450
 baseExponents, 453
 baseScales, 453
 GetIndexForSample(), 453
 kMaxResolution, 453
 offsetForCurrentPixel, 454
 PermutationForDimension(), 452
 pixelForOffset, 454
 radicalInversePermutations, 452
 SampleDimension(), 454
 sampleStride, 453

HenyeyGreenstein, 682
 g, 682
 p(), 682
 Sample_p(), 899
 density, 690

HomogeneousMedium, 689
 g, 689
 Sample(), 893
 sigma_a, 689

sigma_s, 689
sigma_t, 689
Tr(), 689

Hyperboloid, 152

ImageTexture, 619
 ClearCache(), 622
 convertIn(), 622
 convertOut(), 623
 Evaluate(), 623
 GetTexture(), 620
 mapping, 619
 mipmap, 619
 textures, 620

ImageWrap, 626
 Black, 626
 Clamp, 626
 Repeat, 626

InfiniteAreaLight, 737
 distribution, 848
 Le(), 741
 Lmap, 740
 Pdf_Li(), 850
 Power(), 741
 Preprocess(), 740
 Sample_Le(), 959
 Sample_Li(), 849
 worldCenter, 740
 worldRadius, 740

Infinity, 210

Integrator, 25
 Render(), 25

Interaction, 115
 GetMedium(), 688
 Interaction(), 115
 IsMediumInteraction(), 688
 IsSurfaceInteraction(), 116
 mediumInterface, 116
 n, 116
 p, 115
 pError, 115
 SpawnRay(), 232
 SpawnRayTo(), 232
 time, 115
 wo, 115

IntersectionChain, 910
 next, 910
 si, 910

Interval, 112
 high, 112
 low, 112

Inv2Pi, 1063

Inv4Pi, 1063

InvPi, 1063

KdAccelNode, 286
 aboveChild, 286
 AboveChild(), 288

flags, 286
InitInterior(), 288
InitLeaf(), 287

IsLeaf(), 288

nPrimitives(), 288
nPrims, 286
onePrimitive, 286
primitiveIndicesOffset, 286
split, 286
SplitAxis(), 288
SplitPos(), 288

KdSubsurfaceMaterial, 701

KdToDo, 298
 node, 298
 tMax, 298
 tMin, 298

KdTreeAccel, 285
 bounds, 289
 buildTree(), 290
 emptyBonus, 286
 Intersect(), 297
 IntersectP(), 302
 isectCost, 286
 maxPrims, 286
 nAllocatedNodes, 289
 nextFreeNode, 289
 nodes, 289
 primitiveIndices, 287
 primitives, 286
 traversalCost, 286

LambertianReflection, 532
 f(), 533
 R, 532
 rho(), 533

LambertianTransmission, 533

LanczosSincFilter, 481
 Evaluate(), 483
 Sinc(), 483
 tau, 481
 WindowedSinc(), 483

LBVHTreelet, 275
 buildNodes, 275
 nPrimitives, 275
 startIndex, 275

LensElementInterface, 381
 apertureRadius, 381
 curvatureRadius, 381
 eta, 381
 thickness, 381

Light, 714
 flags, 715
 Le(), 741

LightToWorld, 715

mediumInterface, 715
 nSamples, 715
 Pdf_Le(), 955

Pdf_Li(), 836
Power(), 717
Preprocess(), 717
Sample_Le(), 955
Sample_Li(), 716
WorldToLight, 715
LightFlags, 715
 Area, 715
 DeltaDirection, 715
 DeltaPosition, 715
 Infinite, 715
LightStrategy, 852
 UniformSampleAll, 852
 UniformSampleOne, 852
LinearBVHNode, 281
 axis, 281
 bounds, 281
 nPrimitives, 281
 primitivesOffset, 281
 secondChildOffset, 281
LoopSubdiv
 beta(), 196
 faces, 184
 loopGamma(), 203
 vertices, 184
 weightBoundary(), 198
 weightOneRing(), 196
MachineEpsilon, 214
MarbleTexture, 663
 Evaluate(), 664
 mapping, 664
 octaves, 664
 omega, 664
 scale, 664
 variation, 664
Material, 577
 Bump(), 589
 ComputeScatteringFunctions(), 577
Matrix4x4, 1081
 m, 1081
 Mul(), 1081
 Transpose(), 1081
MatteMaterial, 578
 bumpMap, 578
 ComputeScatteringFunctions(), 579
 Kd, 578
 sigma, 578
MaxFloat, 210
MaxMinDistSampler, 465
 CPixel, 466
 StartPixel(), 467
MaxTransforms, 1112
Medium, 684
 Sample(), 891
 Tr(), 684
MediumInteraction, 688
 IsValid(), 893
 MediumInteraction(), 688
 phase, 688
MediumInterface, 684
 inside, 684
 IsMediumTransition(), 685
 outside, 684
MemoryArena, 1074
 Alloc(), 1074
 availableBlocks, 1075
 blockSize, 1074
 currentAllocSize, 1074
 currentBlock, 1074
 currentBlockPos, 1074
 Reset(), 1076
 usedBlocks, 1075
MetalMaterial, 584
MicrofacetDistribution, 537
 D(), 538
 G(), 544
 G1(), 542
 Lambda(), 542
 Pdf(), 811
 sampleVisibleArea, 808
 Sample_wh(), 807
MicrofacetReflection, 547
 distribution, 547
 f(), 547
 fresnel, 547
 Pdf(), 813
 R, 547
 Sample_f(), 811
MicrofacetTransmission, 548
 distribution, 548
 etaA, 548
 etaB, 548
 f(), 548
 fresnel, 548
 mode, 548
 Pdf(), 814
 Sample_f(), 813
 T, 548
MIPMap, 625
 doTrilinear, 625
 EWA(), 637
 Height(), 630
 Levels(), 630
 Lookup(), 635
 Lookup(tri), 632
 maxAnisotropy, 625
 pyramid, 630
 resampleWeights(), 628
 resolution, 625
 Texel(), 631
 triangle(), 634
weightLut, 639
WeightLUTSize, 639
Width(), 630
wrapMode, 625
MirrorMaterial, 584
MitchellFilter, 479
 B, 479
 C, 479
 Evaluate(), 480
 Mitchell1D(), 481
MixMaterial, 582
 ComputeScatteringFunctions(), 582
 m1, 582
 m2, 582
 scale, 582
MixTexture, 616
 amount, 616
 Evaluate(), 617
 tex1, 616
 tex2, 616
MLTIntegrator, 1035
 camera, 1037
 cameraStreamIndex, 1036
 connectionStreamIndex, 1036
 L(), 1036
 largeStepProbability, 1039
 lightStreamIndex, 1036
 maxDepth, 1038
 mutationsPerPixel, 1039
 nBootstrap, 1038
 nChains, 1040
 nSampleStreams, 1036
 Render(), 1038
 sigma, 1039
MLTSampler, 1029
 Accept(), 1031
 currentIteration, 1031
 EnsureReady(), 1032
 Get1D(), 1030
 Get2D(), 1031
 GetNextIndex(), 1035
 largeStep, 1031
 largeStepProbability, 1030
 lastLargeStepIteration, 1031
 Reject(), 1034
 rng, 1030
 sampleIndex, 1035
 sigma, 1030
 StartIteration(), 1031
 StartStream(), 1035
 streamCount, 1030
 streamIndex, 1035
 X, 1030
MortonPrimitive, 271
 mortonCode, 271
 primitiveIndex, 271

NoisePermSize, 652
Normal3, 71
 Normalize(), 71
Normal3f, 71
NumSobolDimensions, 471
OneMinusEpsilon, 417
Options, 1109
 imageFile, 1109
 nThreads, 1109
 quickRender, 1109
 quiet, 1109
 verbose, 1109
OrenNayar, 536
 A, 536
 B, 536
 f(), 536
 R, 536
OrthographicCamera, 361
 dxCamera, 363
 dyCamera, 363
 GenerateRay(), 364
 GenerateRayDifferential(), 364
Paraboloid, 151
ParallelForLoop, 1090
 activeWorkers, 1090
 chunkSize, 1090
 Finished(), 1090
 func1D, 1090
 maxIndex, 1090
 next, 1090
 nextIndex, 1090
 profilerState, 1090
ParamSet, 1105
 AddBlackbodySpectrum(), 1107
 AddBool(), 1107
 AddFloat(), 1107
 AddInt(), 1107
 AddNormal3f(), 1107
 AddPoint2f(), 1107
 AddPoint3f(), 1107
 AddRGBSpectrum(), 1107
 AddSampledSpectrum(), 1107
 AddSampledSpectrumFiles(), 1107
 AddString(), 1107
 AddTexture(), 1107
 AddVector2f(), 1107
 AddVector3f(), 1107
 AddXYZSpectrum(), 1107
 bools, 1105
 Clear(), 1109
 FindBool(), 1108
 FindFloat(), 1108
 FindInt(), 1108
 FindNormal3f(), 1108
 FindOneBool(), 1108
 FindOneFilename(), 1108
FindOneFloat(), 1108
FindOneInt(), 1108
FindOneNormal3f(), 1108
FindOnePoint2f(), 1108
FindOnePoint3f(), 1108
FindOneSpectrum(), 1108
FindOneString(), 1108
FindOneVector2f(), 1108
FindOneVector3f(), 1108
FindPoint2f(), 1108
FindPoint3f(), 1108
 floats, 1105
 ints, 1105
 normals, 1105
 point2fs, 1105
 point3fs, 1105
 ReportUnused(), 1108
 spectra, 1105
 strings, 1105
 textures, 1105
 vector2fs, 1105
 vector3fs, 1105
ParamsetItem, 1106
 lookedUp, 1106
 name, 1106
 nValues, 1106
 values, 1106
PathIntegrator, 875
 Li(), 876
 maxDepth, 876
PbrtOptions, 1109
PerspectiveCamera, 365
 A, 951
 dxCamera, 367
 dyCamera, 367
 GenerateRay(), 367
 GenerateRayDifferential(), 367
 Pdf_We(), 953
 Sample_Wi(), 954
 We(), 950
PhaseFunction, 681
 p(), 681
 Sample_p(), 898
Pi, 1063
PiOver2, 1063
PiOver4, 1063
Pixel, 486
 filterWeightSum, 486
 splatXYZ, 486
 xyz, 486
PixelSampler, 427
 current1DDimension, 427
 current2DDimension, 427
 Get1D(), 428
 Get2D(), 428
 rng, 428
 samples1D, 427
 samples2D, 427
 SetSampleNumber(), 428
 StartNextSample(), 428
PlanarMapping2D, 613
 ds, 613
 dt, 613
 map, 613
 vs, 613
 vt, 613
PlasticMaterial, 580
 bumpMap, 580
 ComputeScatteringFunctions(), 581
 Kd, 580
 Ks, 580
 remapRoughness, 580
 roughness, 580
Point2, 67
 Abs(), 71
 Ceil(), 71
 Floor(), 71
 Lerp(), 70
 Max(), 70
 Min(), 70
 Permute(), 71
 x, 68
 y, 68
Point2f, 68
Point2i, 68
Point3, 68
PerspectiveCamera, 365
 Abs(), 71
 Ceil(), 71
 Floor(), 71
 Lerp(), 70
 Max(), 70
 Min(), 70
 Permute(), 71
 x, 68
 y, 68
 z, 68
Point3f, 68
Point3i, 68
PointLight, 719
 I, 720
 Pdf_Le(), 956
 Pdf_Li(), 836
 pLight, 720
 Power(), 721
 Sample_Le(), 955
 Sample_Li(), 721

PrimarySample, 1030
 Backup(), 1034
 lastModificationIteration, 1032
 modifyBackup, 1034
 Restore(), 1034
 value, 1030
 valueBackup, 1034
Primes, 449
PrimeSums, 452
PrimeTableSize, 449
Primitive, 248
 ComputeScatteringFunctions(), 250
 GetAreaLight(), 249
 GetMaterial(), 249
 Intersect(), 249
 IntersectP(), 249
 WorldBound(), 249
Prof, 1098
ProfilePhase, 1099
 categoryBit, 1099
 reset, 1099
ProfilerState, 1099
ProgressReporter, 1069
 Done(), 1070
 Update(), 1069
ProjectionLight, 724
 cosTotalWidth, 727
 far, 727
 I, 726
 lightProjection, 727
 near, 727
 pLight, 726
 Power(), 728
 Projection(), 727
 projectionMap, 726
 Sample_Li(), 727
 screenBounds, 727
ProjectiveCamera, 358
 CameraToScreen, 360
 focalDistance, 374
 lensRadius, 374
 RasterToCamera, 360
 RasterToScreen, 360
 ScreenToRaster, 360
Quaternion, 99
 Dot(), 100
 Normalize(), 100
 ToTransform(), 101
 v, 100
 w, 100
Ray, 73
 d, 73
 medium, 74
 o, 73
 operator(), 74
 time, 73
 tMax, 73
RayDifferential, 75
 hasDifferentials, 75
 rxDirection, 75
 rxOrigin, 75
 ryDirection, 75
 ryOrigin, 75
 ScaleDifferentials(), 75
RealisticCamera, 378
 BoundExitPupil(), 391
 ComputeCardinalPoints(), 387
 ComputeThickLensApproximation(), 387
 elementInterfaces, 381
 exitPupilBounds, 390
 FocusThickLens(), 388
 GenerateRay(), 394
 IntersectSphericalElement(), 383
 LensFrontZ(), 381
 LensRearZ(), 381
 RearElementRadius(), 382
 SampleExitPupil(), 393
 simpleWeighting, 379
 TraceLensesFromFilm(), 382
 TraceLensesFromScene(), 385
RenderOptions, 1114
 CameraName, 1116
 CameraParams, 1116
 CameraToWorld, 1116
 currentInstance, 1128
 FilterName, 1116
 FilterParams, 1116
 instances, 1128
 lights, 1123
 MakeIntegrator(), 1130
 MakeScene(), 1130
 namedMedia, 1117
 primitives, 1127
 transformEndTime, 1115
 transformStartTime, 1115
ResampleWeight, 627
 firstTexel, 627
 weight, 627
RGB2SpectLambda, 329
RGBIllum2SpectBlue, 329
RGBIllum2SpectCyan, 329
RGBIllum2SpectGreen, 329
RGBIllum2SpectMagenta, 329
RGBIllum2SpectRed, 329
RGBIllum2SpectWhite, 329
RGBIllum2SpectYellow, 329
RGBRef12SpectBlue, 329
RGBRef12SpectCyan, 329
RGBRef12SpectGreen, 329
RGBRef12SpectMagenta, 329
RGBRef12SpectRed, 329
RGBRef12SpectWhite, 329
RGBRef12SpectYellow, 329
 ToRGB(), 328
 ToRGBSpectrum(), 328
 ToXYZ(), 325
 X, 324
 Y, 324
 Z, 324
Sampler, 421
 array1DOffset, 426
 array2DOffset, 426
 Clone(), 424
 currentPixel, 425
 currentPixelSampleIndex, 425
 Get1D(), 422
 Get1DArray(), 424
 Get2D(), 422
 Get2DArray(), 424
 GetCameraSample(), 423
 Request1DArray(), 423
 Request2DArray(), 423
 RoundCount(), 424
 sampleArray1D, 426
 sampleArray2D, 426
 samples1DArraySizes, 426
 samples2DArraySizes, 426

Sampler (continued)

- samplesPerPixel, 422
- SetSampleNumber(), 424
- StartNextSample(), 424
- StartPixel(), 422
- SamplerIntegrator**, 25
 - camera, 26
 - Li(), 31
 - Preprocess(), 26
 - Render(), 26
 - sampler, 25
 - SpecularReflect(), 37
 - SpecularTransmit(), 38
- ScaledBxDF**, 515
 - bxdf, 515
 - f(), 515
 - scale, 515
- ScaleTexture**, 615
 - Evaluate(), 616
 - tex1, 616
 - tex2, 616
- Scene**, 23
 - aggregate, 23
 - Intersect(), 24
 - IntersectP(), 24
 - IntersectTr(), 687
 - lights, 23
 - worldBound, 24
 - WorldBound(), 24
- ScopedAssignment**, 1017
 - backup, 1017
 - ScopedAssignment(), 1018
 - target, 1017
- SDEdge**, 188
 - f, 188
 - f0edgeNum, 188
 - v, 188
- SDFace**, 186
 - children, 186
 - f, 186
 - nextFace(), 192
 - nextVert(), 192
 - otherVert(), 200
 - prevFace(), 192
 - prevVert(), 192
 - v, 186
 - vnum(), 191
- SDVertex**, 185
 - boundary, 185
 - child, 185
 - oneRing(), 196
 - p, 185
 - regular, 185
 - startFace, 185
 - valence(), 191
- SeparableBSSRDF**, 693
 - material, 693
 - mode, 693
 - ns, 693
 - Pdf_Sp(), 912
 - Pdf_Sr(), 909
 - S(), 694
 - Sample_S(), 905
 - Sample_Sp(), 908
 - Sample_Sr(), 909
 - SeparableBSSRDF(), 693
 - Sp(), 695
 - Sr(), 695
 - ss, 693
 - Sw(), 695
 - ts, 693
- SeparableBSSRDFAdapter**, 906
 - bssrdf, 906
 - f(), 906
- ShadowEpsilon**, 232
- Shape**, 123
 - Area(), 131
 - Intersect(), 129
 - IntersectP(), 130
 - ObjectBound(), 125
 - ObjectToWorld, 124
 - Pdf(), 837
 - reverseOrientation, 124
 - Sample(), 837
 - transformSwapsHandedness, 124
 - WorldBound(), 125
 - WorldToObject, 124
- SobolMatrices32**, 471
- SobolMatrixSize**, 471
- SobolSampler**, 468
 - GetIndexForSample(), 470
 - log2Resolution, 470
 - resolution, 470
 - sampleBounds, 470
 - SampleDimension(), 471
- Spectrum**, 315
 - Clamp(), 317
 - Exp(), 317
 - FromRGB(), 330
 - HasNaNs(), 318
 - IsBlack(), 317
 - Lerp(), 317
 - Pow(), 317
 - Sqrt(), 317
 - ToRGB(), 328
 - ToXYZ(), 324
 - y(), 325
- SpectrumType**, 330
 - Illuminant, 330
 - Reflectance, 330
- SpecularReflection**, 524
 - f(), 524
 - fresnel, 524
 - Pdf(), 816
 - R, 524
 - Sample_f(), 525
- SpecularTransmission**, 528
 - etaA, 529
 - etaB, 529
 - f(), 529
 - fresnel, 529
 - mode, 529
 - Pdf(), 816
 - Sample_f(), 529
 - T, 529
- Sphere**, 133
 - Area(), 142
 - Intersect(), 134
 - IntersectP(), 141
 - ObjectBound(), 134
 - Pdf(), 844
 - phiMax, 133
 - radius, 133
 - Sample(), 840
 - thetaMax, 133
 - thetaMin, 133
 - zMax, 133
 - zMin, 133
- SphericalMapping2D**, 611
 - Map(), 611
 - sphere(), 611
 - WorldToTexture, 611
- SplitMethod**, 256
 - EqualCounts, 256
 - HLBVH, 256
 - Middle, 256
 - SAH, 256
- SpotLight**, 721
 - cosFallOffStart, 723
 - cosTotalWidth, 723
 - Falloff(), 724
 - I, 723
 - Pdf_Le(), 956
 - pLight, 723
 - Power(), 724
 - Sample_Le(), 956
 - Sample_Li(), 723
- SPPMIntegrator**, 973
 - camera, 973
 - initialSearchRadius, 973
 - maxDepth, 977
 - nIterations, 975
 - photonsPerIteration, 983
 - Render(), 973
- SPPMPixel**, 974
 - Ld, 978

M, 987
 N, 990
 Phi, 987
 radius, 974
 tau, 990
 vp, 979
SPPMPixelListNode, 980
 next, 980
 pixel, 980
Sqrt2, 1063
StartTransformBits, 1112
StatRegisterer, 1096
 CallCallbacks(), 1097
 funcs, 1096
StatsAccumulator, 1096
 counters, 1096
 ReportCounter(), 1096
StratifiedSampler, 434
 jitterSamples, 434
 StartPixel(), 434
 xPixelSamples, 434
 yPixelSamples, 434
SubstrateMaterial, 584
SubsurfaceMaterial, 700
 bumpMap, 701
 ComputeScatteringFunctions(), 701
 eta, 701
 Kr, 701
 Kt, 701
 remapRoughness, 701
 scale, 701
 sigma_a, 701
 sigma_s, 701
 table, 701
 uRoughness, 701
 vRoughness, 701
SurfaceInteraction, 116
 bsdf, 250
 bssrdf, 250
 ComputeDifferentials(), 601
 ComputeScatteringFunctions(), 578
 dndu, 116
 dndv, 116
 dpdu, 116
 dpdv, 116
 dpdx, 600
 dpdy, 600
 dudx, 600
 dudy, 600
 dvdx, 600
 dvdy, 600
 Le(), 734
 primitive, 249
 SetShadingGeometry(), 119
 shading, 118
 shading::dndu, 118
 shading::dpdu, 118
 shading::dpdv, 118
 shading::n, 118
 shape, 116
 uv, 116
TabulatedBSSRDF, 696
 Pdf_Sr(), 914
 rho, 697
 Sample_Sr(), 914
 sigma_t, 697
 Sr(), 699
 table, 697
TexInfo, 620
Texture, 614
 Evaluate(), 615
TextureMapping2D, 609
 Map(), 610
TextureMapping3D, 614
 Map(), 614
TextureParams, 1120
 FindBool(), 1121
 FindFilename(), 1121
 FindFloat(), 1121
 FindNormal3f(), 1121
 FindPoint3f(), 1121
 FindSpectrum(), 1121
 FindString(), 1121
 FindVector3f(), 1121
 floatTextures, 1120
 geomParams, 1120
 GetSpectrumTexture(), 1121
 materialParams, 1120
 ReportUnused(), 1121
 spectrumTextures, 1120
ThreadIndex, 1089
Transform, 83
 HasScale(), 88
 Inverse(), 85
 IsIdentity(), 85
 m, 84
 mInv, 84
 operator(), 93
 SwapsHandedness(), 96
 Transpose(), 85
TransformCache, 1124
 Lookup(), 1124
TransformedPrimitive, 252
 ComputeScatteringFunctions(), 254
 GetAreaLight(), 254
 Intersect(), 253
 IntersectP(), 253
 Material(), 254
 primitive, 252
 PrimitiveToWorld, 252
 WorldBound(), 254
TransformMapping3D, 614
 Map(), 614
 WorldToTexture, 614
TransformSet, 1112
 Inverse(), 1112
 IsAnimated(), 1114
 t, 1112
TranslucentMaterial, 584
TransportMode, 960
 Importance, 960
 Radiance, 960
Triangle, 156
 Area(), 167
 GetUVs(), 164
 Intersect(), 157
 mesh, 156
 ObjectBound(), 157
 Sample(), 839
 v, 156
 WorldBound(), 157
TriangleFilter, 477
 Evaluate(), 477
TriangleMesh, 154
 alphaMask, 155
 n, 155
 nTriangles, 155
 nVertices, 155
 p, 155
 s, 155
 uv, 155
 vertexIndices, 155
TrowbridgeReitzDistribution, 540
 alphax, 540
 alphay, 540
 D(), 540
 Lambda(), 543
 RoughnessToAlpha(), 540
 Sample_wh(), 811
UberMaterial, 584
UVMapping2D, 610
 du, 610
 dv, 610
 Map(), 610
 su, 610
 sv, 610
UVTexture, 641
 mapping, 641
Vector2, 59
 Abs(), 63
 HasNaNs(), 61
 Max(), 66
 MaxComponent(), 66
 MaxDimension(), 66
 Min(), 66
 MinComponent(), 66
 Permute(), 67

Vector2 (*continued*)
 x, 59
 y, 59
Vector2f, 60
Vector2i, 60
Vector3, 59
 Abs(), 63
 HasNaNs(), 61
 Length(), 65
 LengthSquared(), 65
 Max(), 66
 MaxComponent(), 66
 MaxDimension(), 66
 Min(), 66
 MinComponent(), 66
 Normalize(), 66
 Permute(), 66
 x, 59
 y, 59
 z, 59
Vector3f, 60
Vector3i, 60
Vertex, 996
 beta, 996
 ConvertDensity(), 1000
 CreateCamera(), 997
 CreateLight(), 997
 CreateMedium(), 997
 CreateSurface(), 997
 delta, 998
 ei, 996
 f(), 998
 GetInteraction(), 997
 IsConnectible(), 998
 IsDeltaLight(), 999
 IsInfiniteLight(), 999
 IsLight(), 999
 IsOnSurface(), 998
 Le(), 999
 mi, 996
 ng(), 997
 ns(), 997
 p(), 997
 Pdf(), 1001
 pdfFwd, 1000
 PdfLight(), 1002
 PdfLightOrigin(), 1003
 pdfRev, 1000
 si, 996
 time(), 997
 type, 996
VertexType, 996
 Camera, 996
 Light, 996
 Medium, 996
 Surface, 996
VisibilityTester, 717
 p0, 717
 P0(), 718
 p1, 717
 P1(), 718
 Tr(), 718
 Unoccluded(), 718
VisiblePoint, 979
 beta, 979
 bsdf, 979
 p, 979
 wo, 979
VolPathIntegrator, 900
 Li(), 900
WhittedIntegrator, 32
 Li(), 33
 maxDepth, 33
WindyTexture, 662
 Evaluate(), 663
 mapping, 662
WrinkledTexture, 662
ZeroTwoSequenceSampler, 462
 RoundCount(), 462
 StartPixel(), 463

INDEX OF MISCELLANEOUS IDENTIFIERS

Finally, this index covers functions, module-local variables, preprocessor definitions, and other miscellaneous identifiers used in the system.

AbsCosTheta()	510	ComputeLightPowerDistribution()	72
AbsDot()	64	Faceforward()	72
activeTransformBits	1111	FBm()	656
alloca()	1070	FindInterval()	1065
AllocAligned()	1072	FloatToBits()	211
ARENA_ALLOC()	576	FOR_ACTIVE_TRANSFORMS()	1112
Assert()	1069	Fourier()	559
AverageSpectrumSamples()	321	FrConductor()	521
BalanceHeuristic()	799	FrDielectric()	519
BeamDiffusionMS()	928	FreeAligned()	1072
BeamDiffusionSS()	934	FresnelMoment1()	695
BitsToFloat()	212	FresnelMoment2()	695
Blackbody()	710	G()	1011
BlackbodyNormalized()	711	gamma()	217
BlossomBezier()	172	GammaCorrect()	621
BSDF_ALL	513	Gaussian()	479
BSDF_DIFFUSE	513	GenerateCameraSubpath()	1003
BSDF_GLOSSY	513	GenerateLightSubpath()	1004
BSDF_REFLECTION	513	GetMediumScatteringProperties()	702
BSDF_SPECULAR	513	Grad()	652
BSDF_TRANSMISSION	513	graphicsState	1119
BSSRDF	692	GrayCode()	459
CatmullRomWeights()	562	GrayCodeSample()	461
CIE_lambda	323	hash()	982
CIE_X	323	ImageWrap	626
CIE_Y	323	InfiniteLightDensity()	1021
CIE_Y_integral	325	Info()	1068
CIE_Z	323	InitProfiler()	1100
Clamp()	1062	IntegrateCatmullRom()	937
ComputeBeamDiffusionBSSRDF()	935	InterpolateSpectrumSamples()	333

IntervalFindZeros(), 113
 Inverse(), 1081
 InverseGammaCorrect(), 622
 InverseRadicalInverse(), 448
 InvertCatmullRom(), 939
 IsDeltalight(), 715
 IsPowerOf2(), 1064
 Lanczos(), 629
 LatinHypercube(), 440
 LeftShift3(), 272
 Lerp(), 1079
 Log2(), 1063
 Log2Int(), 1064
 LookAt(), 92
 LoopSubdivide(), 184
 main(), 21
 MakeLight(), 1131
 MakeShapes(), 1125
 MaxThreadIndex(), 1093
 MISWeight(), 1016
 Mod(), 1062
 MultiplyGenerator(), 458
 namedCoordinateSystems, 1113
 nCIESamples, 323
 NEXT(), 186
 NextFloatDown(), 213
 NextFloatUp(), 212
 Noise(), 649
 NoisePerm, 652
 NoiseWeight(), 653
 nRGB2SpectSamples, 329
 nSpectralSamples, 319
 nSpectrumSamples, 316
 NumSystemCores(), 1088
 OffsetRayOrigin(), 231
 Orthographic(), 363
 ParallelFor(), 1088
 ParallelFor2D(), 1093
 ParseFile(), 21
 PBRT_L1_CACHE_LINE_SIZE, 1072
 pbrtAccelerator(), 1116
 pbrtActiveTransformAll(), 1115
 pbrtActiveTransformEndTime(), 1115
 pbrtActiveTransformStartTime(), 1115
 pbrtAreaLightSource(), 1123
 pbrtAttributeBegin(), 1119
 pbrtAttributeEnd(), 1119
 pbrtCamera(), 1116
 pbrtCleanup(), 1109
 pbrtConcatTransform(), 1113
 pbrtCoordinateSystem(), 1113
 pbrtCoordSysTransform(), 1113
 pbrtFilm(), 1116
 pbrtIdentity(), 1112
 pbrtInit(), 1109
 pbrtIntegrator(), 1116
 pbrtLightSource(), 1123
 pbrtLookAt(), 1113
 pbrtMakeNamedMaterial(), 1122
 pbrtMakeNamedMedium(), 1117
 pbrtMaterial(), 1122
 pbrtMediumInterface(), 1117
 pbrtNamedMaterial(), 1122
 pbrtObjectBegin(), 1128
 pbrtObjectEnd(), 1128
 pbrtObjectInstance(), 1128
 pbrtPixelFilter(), 1115
 pbrtRotate(), 1113
 pbrtSampler(), 1116
 pbrtScale(), 1113
 pbrtShape(), 1124
 pbrtTexture(), 1121
 pbrtTransform(), 1113
 pbrtTransformBegin(), 1119
 pbrtTransformEnd(), 1119
 pbrtTransformTimes(), 1115
 pbrtTranslate(), 1113
 pbrtWorldBegin(), 1117
 pbrtWorldEnd(), 1129
 Perspective(), 365
 PhaseHG(), 681
 PowerHeuristic(), 799
 PREV(), 186
 PrintStats(), 1097
 profileSamples, 1100
 pushedActiveTransformBits, 1119
 pushedGraphicsStates, 1119
 pushedTransforms, 1119
 Quadratic(), 1079
 Radians(), 1063
 RadicalInverse(), 444
 RadicalInverseSpecialized(), 447
 RadixSort(), 274
 RandomWalk(), 1005
 ReadFloatFile(), 1070
 ReadImage(), 1067
 Reflect(), 526
 Refract(), 531
 RejectionSampleDisk(), 762
 renderOptions, 1114
 ReportProfilerResults(), 1100
 ReportProfileSample(), 1100
 ReportThreadStats(), 1097
 ReportValue(), 1094
 ReverseBits32(), 446
 ReverseBits64(), 446
 RGBToXYZ(), 328
 RNG, 1065
 Rotate(), 91
 RotateX(), 89
 RotateY(), 89
 RotateZ(), 89
 RoundUpPow2(), 1064
 SameHemisphere(), 807
 SampleCatmullRom(), 823
 SampleCatmullRom2D(), 824
 sampledLambdaEnd, 319
 sampledLambdaStart, 319
 SampleFourier(), 828
 SampleGeneratorMatrix(), 459
 Scale(), 87
 ScrambledRadicalInverse(), 450
 Severe(), 1068
 Shuffle(), 438
 shutdownThreads, 1088
 Sin2Phi(), 511
 Sin2Theta(), 510
 SinPhi(), 511
 SinTheta(), 510
 Slerp(), 103
 SmoothStep(), 658
 Sobol2D(), 464
 SobolIntervalToIndex(), 470
 SobolSample(), 471
 SobolSampleDouble(), 471
 SobolSampleFloat(), 471
 SolveLinearSystem2x2(), 1080
 SortSpectrumSamples(), 320
 SpectrumSamplesSorted(), 319
 SphericalDirection(), 346
 SphericalPhi(), 346
 SphericalTheta(), 346
 statsAccumulator, 1097
 StratifiedSample1D(), 437
 StratifiedSample2D(), 437
 SubdivideBezier(), 175
 SubsurfaceFromDiffuse(), 938
 Tan2Theta(), 510
 TanTheta(), 510
 TerminateWorkerThreads(), 1093
 threads, 1088
 ToGrid(), 982
 transformCache, 1125
 Translate(), 87
 Turbulence(), 660
 UniformConePdf(), 781
 UniformHemispherePdf(), 775
 UniformSampleAllLights(), 854
 UniformSampleCone(), 781
 UniformSampleDisk(), 777
 UniformSampleHemisphere(), 775
 UniformSampleOneLight(), 856
 UniformSampleSphere(), 776
 UniformSampleTriangle(), 782
 UniformSpherePdf(), 776
 VanDerCorput(), 463
 VERIFY_INITIALIZED, 1111

VERIFY_OPTIONS(), 1111
VERIFY_WORLD(), 1111
WARN_IF_ANIMATED_TRANSFORM(), 1114
Warning(), 1068
workerThreadFunc(), 1092
workList, 1089
workListCondition, 1090
workListMutex, 1089
WriteImage(), 1068
XYZToRGB(), 327

This page intentionally left blank

References

- Acton, M. 2014. Data-oriented design and C++. <http://www.slideshare.net/cellperformance/data-oriented-design-and-c>.
- Adams, A., and M. Levoy. 2007. General linear cameras with finite aperture. In *Proceedings of the 2007 Eurographics Symposium on Rendering*, 121–26.
- Áfra, A. 2012. Incoherent ray tracing without acceleration structures. *Eurographics 2012 Short Paper*.
- Akalin, F. 2015. A better way to sample a sphere (w.r.t. solid angle). <https://www.akalin.com/sampling-visible-sphere>.
- Aila, T., and T. Karras. 2010. Architecture considerations for tracing incoherent rays. In *Proceedings of High Performance Graphics 2010*, 113–22.
- Aila, T., T. Karras, and S. Laine. 2013. On quality metrics of bounding volume hierarchies. In *Proceedings of High Performance Graphics 2013*, 101–07.
- Aila, T., and S. Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of High Performance Graphics 2009*, 145–50.
- Akenine-Möller, T. 2001. Fast 3D triangle-box overlap testing. *Journal of Graphics Tools* 6(1), 29–33.
- Akenine-Möller, T., E. Haines, and N. Hoffman. 2008. *Real-Time Rendering*. Natick, MA: A. K. Peters.
- Akenine-Möller, T., and J. Hughes. 1999. Efficiently building a matrix to rotate one vector to another. *Journal of Graphics Tools* 4(4), 1–4.
- Alim, U. R. 2013. Rendering in shift-invariant spaces. In *Proceedings of Graphics Interface 2013*, 189–96.
- Amanatides, J. 1984. Ray tracing with cones. *Computer Graphics (SIGGRAPH '84 Proceedings)*, 18, 129–35.
- Amanatides, J. 1992. Algorithms for the detection and elimination of specular aliasing. In *Proceedings of Graphics Interface 1992*, 86–93.
- Amanatides, J., and D. P. Mitchell. 1990. Some regularization problems in ray tracing. In *Proceedings of Graphics Interface 1990*, 221–28.
- Amanatides, J., and A. Woo. 1987. A fast voxel traversal algorithm for ray tracing. In *Proceedings of Eurographics '87*, 3–10.
- Ament, M., C. Bergmann, and D. Weiskopf. 2014. Refractive radiative transfer equation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014)* 33(2), 17:1–17:22.
- Anderson, S. 2004. graphics.stanford.edu/~seander/bithacks.html.
- Anton, H. A., I. Bivens, and S. Davis. 2001. *Calculus* (7th ed.). New York: John Wiley & Sons.
- Apodaca, A. A., and L. Gritz. 2000. *Advanced RenderMan: Creating CGI for Motion Pictures*. San Francisco: Morgan Kaufmann.

- Appel, A. 1968. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conference*, 32, 37–45.
- Arbree, A., B. Walter, and K. Bala. 2011. Heterogeneous subsurface scattering using the finite element method. *IEEE Transactions on Visualization and Computer Graphics* 17(7), 956–69.
- Arnaldi, B., T. Priol, and K. Bouatouch. 1987. A new space subdivision method for ray tracing CSG modeled scenes. *The Visual Computer* 3(2), 98–108.
- Arvo, J. 1986. Backward ray tracing. *Developments in Ray Tracing, SIGGRAPH '86 Course Notes*.
- Arvo, J. 1988. Linear-time voxel walking for octrees. *Ray Tracing News* 12(1).
- Arvo, J. 1990. Transforming axis-aligned bounding boxes. In A. S. Glassner (Ed.), *Graphics Gems I*, 548–50. San Diego: Academic Press.
- Arvo, J. 1993. Transfer equations in global illumination. In *Global Illumination, SIGGRAPH '93 Course Notes*, Volume 42.
- Arvo, J. 1995a. Analytic methods for simulated light transport. Ph.D. thesis, Yale University.
- Arvo, J. 1995b. Stratified sampling of spherical triangles. In *Proceedings of SIGGRAPH 1995*, 437–38.
- Arvo, J., and D. Kirk. 1987. Fast ray tracing by ray classification. *Computer Graphics (SIGGRAPH '87 Proceedings)* 21(4), 55–64.
- Arvo, J., and D. Kirk. 1990. Particle transport and image synthesis. *Computer Graphics (SIGGRAPH '90 Proceedings)* 24(4), 63–66.
- Ashdown, I. 1993. Near-field photometry: a new approach. *Journal of the Illuminating Engineering Society* 22(1), 163–80.
- Ashdown, I. 1994. *Radiosity: A Programmer's Perspective*. New York: John Wiley & Sons.
- Ashikhmin, M., and P. Shirley. 2000. An anisotropic Phong light reflection model. *Technical Report UU-CS-00-014*. University of Utah.
- Ashikhmin, M., and P. Shirley. 2002. An anisotropic Phong BRDF model. *Journal of Graphics Tools* 5(2), 25–32.
- Ashikhmin, M., S. Premoze, and P. S. Shirley. 2000. A microfacet-based BRDF generator. In *Proceedings of ACM SIGGRAPH 2000*, 65–74.
- Atcheson, B., I. Ihrke, W. Heidrich, A. Tevs, D. Bradley, M. Magnor, and H.-P. Seidel. 2008. Time-resolved 3d capture of non-stationary gas flows. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 27(5), 132:1–132:9.
- Atkinson, K. 1993. *Elementary Numerical Analysis*. New York: John Wiley & Sons.
- Badouel, D., and T. Priol. 1989. An efficient parallel ray tracing scheme for highly parallel architectures. In *Fifth Eurographics Workshop on Graphics Hardware*.
- Bagher, M., C. Soler, N. Holzschuch. 2012. Accurate fitting of measured reflectances using a shifted gamma micro-facet distribution. *Computer Graphics Forum* 31(4), 1509–18.
- Bahar, E., and S. Chakrabarti. 1987. Full-wave theory applied to computer-aided graphics for 3D objects. *IEEE Computer Graphics and Applications* 7(7), 46–60.
- Banks, D. C. 1994. Illumination in diverse codimensions. In *Proceedings of SIGGRAPH '94*, Computer Graphics Proceedings, Annual Conference Series, 327–34.
- Barkans, A. C. 1997. High-quality rendering using the Talisman architecture. In *1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 79–88.

- Barringer, R., and T. Akenine-Möller. 2014. Dynamic ray stream traversal. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014)* 33(4), 151:1–151:9.
- Barzel, R. 1997. Lighting controls for computer cinematography. *Journal of Graphics Tools* 2(1), 1–20.
- Bashford-Rogers, T., K. Debattista, and A. Chalmers. 2013. Importance driven environment map sampling. *IEEE Transactions on Visualization and Computer Graphics* 20(6), 907–18.
- Bauszat, P., M. Eisemann, E. Eisemann, and M. Magnor. 2015. General and robust error estimation and reconstruction for Monte Carlo rendering. *Computer Graphics Forum (Proceedings of Eurographics 2015)* 34(2), 597–608.
- Bauszat, P., M. Eisemann, and M. Magnor. 2010. The minimal bounding volume hierarchy. *Vision, Modeling, and Visualization (2010)*.
- Becker, B. G., and N. L. Max. 1993. Smooth transitions between bump rendering algorithms. In *Proceedings of SIGGRAPH '93, Computer Graphics Proceedings, Annual Conference Series*, 183–90.
- Beckmann, P., and A. Spizzichino. 1963. *The Scattering of Electromagnetic Waves from Rough Surfaces*. New York: Pergamon.
- Belcour, L., C. Soler, K. Subr, N. Holzschuch, and F. Durand. 2013. 5D covariance tracing for efficient defocus and motion blur. *ACM Transactions on Graphics* 32(3), 31:1–31:18.
- Benthin, C. 2006. Realtime ray tracing on current CPU architectures. Ph.D. thesis, Saarland University.
- Benthin, C., and I. Wald. 2009. Efficient ray traced soft shadows using multi-frusta tracing. In *Proceedings of High Performance Graphics 2009*, 135–44.
- Benthin, C., S. Boulos, D. Lacewell, and I. Wald. 2007. Packet-based ray tracing of Catmull–Clark subdivision surfaces. *SCI Institute Technical Report, No. UUSCI-2007-011*. University of Utah.
- Benthin, C., I. Wald, and P. Slusallek. 2003. A scalable approach to interactive global illumination. In *Computer Graphics Forum* 22(3), 621–30.
- Benthin, C., I. Wald, and P. Slusallek. 2004. Techniques for interactive ray tracing of Bézier surfaces. *Journal of Graphics, GPU, and Game Tools* 11(2), 1–16.
- Benthin, C., I. Wald, S. Woop, M. Ernst, and W. R. Mark. 2011. Combining single and packet ray tracing for arbitrary ray distributions on the Intel(r) MIC architecture. *IEEE Transactions on Visualization and Computer Graphics* 18(9), 1438–48.
- Berger, E. D., B. G. Zorn, and K. S. McKinley. 2001. Composing high-performance memory allocators. In *SIGPLAN Conference on Programming Language Design and Implementation*, 114–24.
- Berger, E. D., B. G. Zorn, and K. S. McKinley. 2002. Reconsidering custom memory allocation. In *Proceedings of ACM OOPSLA 2002*.
- Betrisey, C., J. F. Blinn, B. Dresevic, B. Hill, G. Hitchcock, B. Keely, D. P. Mitchell, J. C. Platt, and T. Whitted. 2000. Displaced filtering for patterned displays. *Society for Information Display International Symposium. Digest of Technical Papers* 31, 296–99.
- Bhate, N., and A. Tokuta. 1992. Photorealistic volume rendering of media with directional scattering. In *Proceedings of the Third Eurographics Rendering Workshop*, 227–45.
- Bigler, J., A. Stephens, and S. Parker. 2006. Design for parallel interactive ray tracing systems. *IEEE Symposium on Interactive Ray Tracing*, 187–95.

- Billen, N., B. Engelen, A. Lagae, and P. Dutré. 2013. Probabilistic visibility evaluation for direct illumination. *Computer Graphics Forum (Proceedings of the 2013 Eurographics Symposium on Rendering)* 32(4), 39–47.
- Billen, N., A. Lagae, and P. Dutré. 2014. Probabilistic visibility evaluation using geometry proxies. *Computer Graphics Forum (Proceedings of the 2014 Eurographics Symposium on Rendering)* 33(4), 143–52.
- Bitterli, B., J. Novák, and W. Jarosz. 2015. Portal-masked environment map sampling. *Computer Graphics Forum (Proceedings of the 2015 Eurographics Symposium on Rendering)* 34(4).
- Bittner, J., M. Hapala, and V. Havran. 2013. Fast insertion-based optimization of bounding volume hierarchies. *Computer Graphics Forum* 32(1), 85–100.
- Bittner, J., M. Hapala, and V. Havran. 2014. Incremental BVH construction for ray tracing. *Computers & Graphics* 47, 135–44.
- Bjorke, K. 2001. Using Maya with RenderMan on Final Fantasy: The Spirits Within. *SIGGRAPH 2001 RenderMan Course Notes*.
- Blasi, P., B. L. Saëc, and C. Schlick. 1993. A rendering algorithm for discrete volume density objects. *Computer Graphics Forum (Proceedings of Eurographics '93)* 12(3), 201–10.
- Blinn, J. F. 1977. Models of light reflection for computer synthesized pictures. *Computer Graphics (SIGGRAPH '77 Proceedings)*, 11, 192–98.
- Blinn, J. F. 1978. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, 12, 286–92.
- Blinn, J. F. 1982a. A generalization of algebraic surface drawing. *ACM Transactions on Graphics* 1(3), 235–56.
- Blinn, J. F. 1982b. Light reflection functions for simulation of clouds and dusty surfaces. *Computer Graphics* 16(3), 21–29.
- Blinn, J. F., and M. E. Newell. 1976. Texture and reflection in computer generated images. *Communications of the ACM* 19, 542–46.
- Bloom, C., J. Blow, and C. Muratori. 2004. Errors and omissions in Marc Alexa's "Linear combination of transformations." www.cbloom.com/3d/techdocs/lcot_errors.pdf
- Blow, J. 2004. Understanding slerp, then not using it. *Game Developer Magazine*. Also available from number-none.com/product/Understanding_Slerp_Then_Not_Using_It
- Blumofe, R., and C. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM* 46(5), 720–48.
- Blumofe, R., C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. 1996. Cilk: an efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing* 37(1), 55–69.
- Boehm, H.-J. 2005. Threads cannot be implemented as a library. *ACM SIGPLAN Notices* 40(6), 261–68.
- Bolz, J., and P. Schröder. 2002. Rapid evaluation of Catmull–Clark subdivision surfaces. In *Web3D 2002 Symposium*.
- Booth, T. E. 1986. A Monte Carlo learning/biasing experiment with intelligent random numbers. *Nuclear Science and Engineering* 92, 465–81.
- Borges, C. 1991. Trichromatic approximation for computer graphics illumination models. In *Computer Graphics (Proceedings of SIGGRAPH '91)*, 25, 101–04.

- Boulos, S., and E. Haines. 2006. Ray–box sorting. *Ray Tracing News* 19(1), tog.acm.org/resources/RTNews/html/rtnv19n1.html.
- Boulos, S., I. Wald, and C. Benthin. 2008. Adaptive ray packet reordering. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, 131–38.
- Bracewell, R. N. 2000. *The Fourier Transform and Its Applications*. New York: McGraw-Hill.
- Brady, A., J. Lawrence, P. Peers, and W. Weimer. 2014. genBRDF: discovering new analytic BRDFs with genetic programming. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014)* 33(4), 114:1–114:11.
- Bronsvort, W. F., and F. Klok. 1985. Ray tracing generalized cylinders. *ACM Transactions on Graphics* 4(4), 291–303.
- Bruneton, E., and F. Neyret. 2012. A survey of nonlinear prefiltering methods for efficient and accurate surface shading. *IEEE Transactions on Visualization and Computer Graphics* 18(2), 242–60.
- Buck, R. C. 1978. *Advanced Calculus*. New York: McGraw-Hill.
- Budge, B., D. Coming, D. Norpchen, and K. Joy. 2008. Accelerated building and ray tracing of restricted BSP trees. In *IEEE Symposium on Interactive Ray Tracing*, 167–74.
- Budge, B., T. Bernardin, J. Stuart, S. Sengupta, K. Joy, and J. D. Owens. 2009. Out-of-core data management for path tracing on hybrid resources. *Computer Graphics Forum (Proceedings of Eurographics 2009)* 28(2), 385–96.
- Buhler, J., and D. Wexler. 2002. A phenomenological model for Bokeh rendering. *SIGGRAPH 2002 Sketch*.
- Burke, D., A. Ghosh, and W. Heidrich. 2005. Bidirectional importance sampling for direct illumination. In *Rendering Techniques 2005: 16th Eurographics Workshop on Rendering*, 147–56.
- Burley, B. 2012. Physically-based shading at Disney. *Physically Based Shading in Film and Game Production, SIGGRAPH 2012 Course Notes*.
- Buss, S., and J. Fillmore. 2001. Spherical averages and applications to spherical splines and interpolation. *ACM Transactions on Graphics* 20(2), 95–126.
- Cabral, B., N. Max, and R. Springmeyer. 1987. Bidirectional reflection functions from surface bump maps. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21, 273–81.
- Calder, B., K. Chandra, S. John, and T. Austin. 1998. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose.
- Cant, R. J., and P. A. Shrubsole. 2000. Texture potential MIP mapping, a new high-quality texture antialiasing algorithm. *ACM Transactions on Graphics* 19(3), 164–84.
- Carr, N., J. D. Hall, and J. Hart. 2002. The ray engine. In *Proceedings of Graphics Hardware 2002*.
- Catmull, E., and J. Clark. 1978. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design* 10, 350–55.
- Cazals, F., G. Drettakis, and C. Puech. 1995. Filtering, clustering and hierarchy construction: a new solution for ray-tracing complex scenes. *Computer Graphics Forum* 14(3), 371–82.
- Cerezo, E., F. Perez-Cazorla, X. Pueyo, F. Seron, and F. Sillion. 2005. A survey on participating media rendering techniques. *The Visual Computer* 21(5), 303–28.

- Chandrasekhar, S. 1960. *Radiative Transfer*. New York: Dover Publications. Originally published by Oxford University Press, 1950.
- Chen, J., K. Venkataraman, D. Bakin, B. Rodricks, R. Gravelle, P. Rao, and Y. Ni. 2009. Digital camera imaging system simulation. *IEEE Transactions on Electron Devices* 56(11), 2496–05.
- Chen, J., B. Wang, and J.-H. Yong. 2011. Improved stochastic progressive photon mapping with Metropolis sampling. *Computer Graphics Forum (Proceedings of the 2011 Eurographics Symposium on Rendering)* 30(4), 1205–13.
- Chib, S., and E. Greenberg. 1995. Understanding the Metropolis–Hastings algorithm. *The American Statistician* 49(4), 327–35.
- Chilimbi, T. M., B. Davidson, and J. R. Larus. 1999a. Cache-conscious structure definition. In *SIGPLAN Conference on Programming Language Design and Implementation*, 13–24.
- Chilimbi, T. M., M. D. Hill, and J. R. Larus. 1999b. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1–12.
- Chiu, K., P. Shirley, and C. Wang. 1994. Multi-jittered sampling. In P. Heckbert (Ed.), *Graphics Gems IV*, 370–74. San Diego: Academic Press.
- Choi, B., R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart. 2010. Parallel SAH k-D tree construction. In *Proceedings of High Performance Graphics 2010*, 77–86.
- Choi, B., B. Chang, and I. Ihm. 2013. Improving memory space efficiency of kd-tree for real-time ray tracing. *Computer Graphics Forum* 32(7), 335–44.
- Christensen, P. H. 2003. Adjoints and importance in rendering: an overview. *IEEE Transactions on Visualization and Computer Graphics* 9(3), 329–40.
- Christensen, P. 2015. The path-tracing revolution in the movie industry. *SIGGRAPH 2015 Course*.
- Christensen, P. H., and B. Burley. 2015. Approximate reflectance profiles for efficient subsurface scattering. *Pixar Technical Memo 15-04*.
- Christensen, P. H., D. M. Laur, J. Fong, W. L. Wooten, and D. Batali. 2003. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. In *Computer Graphics Forum (Eurographics 2003 Conference Proceedings)* 22(3), 543–52.
- Chvolson, O. D. 1890. Grundzüge einer mathematischen Theorie der inneren Diffusion des Lichtes. *Izv. Peterburg. Academii Nauk* 33, 221–65.
- CIE Technical Report. 2004. Colorimetry. *Publication 15:2004 (3rd ed.)*, CIE Central Bureau, Vienna.
- Clarberg, P., and T. Akenine-Möller. 2008a. Practical product importance sampling for direct illumination. *Computer Graphics Forum (Proceedings of Eurographics 2008)* 27(2), 681–90.
- Clarberg, P., and T. Akenine-Möller. 2008b. Exploiting visibility correlation in direct illumination. *Computer Graphics Forum (Proceedings of the 2008 Eurographics Symposium on Rendering)* 27(4), 1125–36.
- Clarberg, P., W. Jarosz, T. Akenine-Möller, and H. W. Jensen. 2005. Wavelet importance sampling: efficiently evaluating products of complex functions. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)* 24(3), 1166–75.
- Clark, J. H. 1976. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM* 19(10), 547–54.
- Cleary, J. G., and G. Wyvill. 1988. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer* 4(2), 65–83.

- Cleary, J. G., B. M. Wyvill, R. Vatti, and G. M. Birtwistle. 1983. Design and analysis of a parallel ray tracing computer. In *Proceedings of Graphics Interface 1983*, 33–38.
- Cline, D., D. Adams, and P. Egbert. 2008. Table-driven adaptive importance sampling. *Computer Graphics Forum (Proceedings of the 2008 Eurographics Symposium on Rendering)* 27(4), 1115–23.
- Cline, D., J. Talbot, and P. Egbert. 2005. Energy redistribution path tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)* 24(3), 1186–95.
- Cline, D., P. Egbert, J. Talbot, and D. Cardon. 2006. Two stage importance sampling for direct lighting. *Rendering Techniques 2006: 17th Eurographics Workshop on Rendering*, 103–14.
- Cline, D., A. Razdan, and P. Wonka. 2009. A comparison of tabular PDF inversion methods. *Computer Graphics Forum* 28(1), 154–60.
- Cohen, J., A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. P. Brooks Jr., and W. Wright. 1996. Simplification envelopes. In *Proceedings of SIGGRAPH '96*, Computer Graphics Proceedings, Annual Conference Series, 119–28.
- Cohen, M., and D. P. Greenberg. 1985. The hemi-cube: a radiosity solution for complex environments. *SIGGRAPH Computer Graphics* 19(3), 31–40.
- Cohen, M., and J. Wallace. 1993. *Radiosity and Realistic Image Synthesis*. San Diego: Academic Press Professional.
- Collins, S. 1994. Adaptive splatting for specular to diffuse light transport. In *Fifth Eurographics Workshop on Rendering*, Darmstadt, Germany, 119–35.
- Contini, D., F. Martelli, and G. Zaccanti. 1997. Photon migration through a turbid slab described by a model based on diffusion approximation. I. Theory. *Applied Optics* 36(19), 4587–4599.
- Cook, R. L. 1984. Shade trees. *Computer Graphics (SIGGRAPH '84 Proceedings)*, 18, 223–31.
- Cook, R. L. 1986. Stochastic sampling in computer graphics. *ACM Transactions on Graphics* 5(1), 51–72.
- Cook, R., and T. DeRose. 2005. Wavelet noise. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)* 24(3), 803–11.
- Cook, R. L., and K. E. Torrance. 1981. A reflectance model for computer graphics. *Computer Graphics (SIGGRAPH '81 Proceedings)*, 15, 307–16.
- Cook, R. L., and K. E. Torrance. 1982. A reflectance model for computer graphics. *ACM Transactions on Graphics* 1(1), 7–24.
- Cook, R. L., T. Porter, and L. Carpenter. 1984. Distributed ray tracing. *Computer Graphics (SIGGRAPH '84 Proceedings)*, 18, 137–45.
- Cook, R. L., L. Carpenter, and E. Catmull. 1987. The Reyes image rendering architecture. *Computer Graphics (Proceedings of SIGGRAPH '87)*, 95–102.
- Crow, F. C. 1977. The aliasing problem in computer-generated shaded images. *Communications of the ACM* 20(11), 799–805.
- Crow, F. C. 1984. Summed-area tables for texture mapping. *Computer Graphics (Proceedings of SIGGRAPH '84)*, 18, 207–12.
- Cuypers, T., T. Haber, P. Bekaert, S. B. Oh, and R. Raskar. 2012. Reflectance model for diffraction. *ACM Transactions on Graphics* 31(5), 122:1–122:11.
- Dachsbacher, C., J. Křivánek, M. Hašan, A. Arbree, B. Walter, and J. Novák. 2014. Scalable realistic rendering with many-light methods. *Computer Graphics Forum* 33(1), 88–104.

- Dammertz, H., and J. Hanika. 2009. Plane sampling for light paths from the environment map. *journal of graphics, gpu, and game tools* 14(2), 25–31.
- Dammertz, H., J. Hanika, and A. Keller. 2008. Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. *Computer Graphics Forum* 27(4), 1225–33.
- Dammertz, H., and A. Keller. 2006. Improving ray tracing precision by object space intersection computation. *IEEE Symposium on Interactive Ray Tracing*, 25–31.
- Dammertz, H., and A. Keller. 2008a. The edge volume heuristic—robust triangle subdivision for improved BVH performance, In *IEEE Symposium on Interactive Ray Tracing*, 155–58.
- Dammertz, S., and A. Keller. 2008b. Image synthesis by rank-1 lattices. *Monte Carlo and Quasi-Monte Carlo Methods 2006*, 217–36.
- Dana, K. J., B. van Ginneken, S. K. Nayar, and J. J. Koenderink. 1999. Reflectance and texture of real-world surfaces. *ACM Transactions on Graphics* 18(1), 1–34.
- Danskin, J., and P. Hanrahan. 1992. Fast algorithms for volume ray tracing. In *1992 Workshop on Volume Visualization*, 91–98.
- Daumas, M., and G. Melquiond. 2010. Certification of bounds on expressions involving rounded operators. *ACM Transactions on Mathematical Software* 37(1), 2:1–2:20.
- Davidovič, T., J. Křivánek, M. Hašan, and P. Slusallek. 2014. Progressive light transport simulation on the GPU: survey and improvements. *ACM Transactions on Graphics* 33(3), 29:1–29:19.
- de Goes, F., K. Breeden, V. Ostromoukhov, and M. Desbrun. 2012. Blue noise through optimal transport. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 31(6), 171:1–171:11.
- de Voogt, E., A. van der Helm, and W. F. Bronsvoort. 2000. Ray tracing deformed generalized cylinders. *The Visual Computer* 16(3–4), 197–207.
- Debevec, P. 1998. Rendering synthetic objects into real scenes: bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Proceedings of SIGGRAPH '98*, 189–98.
- Deering, M. F. 1995. Geometry compression. In *Proceedings of SIGGRAPH '95*, Computer Graphics Proceedings, Annual Conference Series, 13–20.
- d’Eon, E., G. Francois., M. Hill, J. Letteri, and J.-M. Aubry. 2011. An energy-conserving hair reflectance model. *Computer Graphics Forum* 30(4), 1181–87.
- d’Eon, E., and G. Irving. 2011. A quantized-diffusion model for rendering translucent materials. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2011)* 28(3), 56:1–56:14.
- d’Eon, E., D. Luebke, and E. Enderton. 2007. Efficient rendering of human skin. In *Rendering Techniques 2007: 18th Eurographics Workshop on Rendering*, 147–58.
- d’Eon, E., S. Marschner, and J. Hanika. 2013. Importance sampling for physically-based hair fiber models. In *SIGGRAPH Asia 2013 Technical Briefs*, 25:1–25:4.
- Delbracio, M., P. Musé, A. Buades, J. Chauvier, N. Phelps, and J.-M. Morel. 2014. Boosting Monte Carlo rendering by ray histogram fusion. *ACM Transactions on Graphics* 33(1), 8:1–8:15.
- DeRose, T. D. 1989. *A Coordinate-Free Approach to Geometric Programming. Math for SIGGRAPH, SIGGRAPH Course Notes #23*. Also available as Technical Report No. 89-09-16, Department of Computer Science and Engineering, University of Washington, Seattle.

- Deussen, O., P. M. Hanrahan, B. Lintermann, R. Mech, M. Pharr, and P. Prusinkiewicz. 1998. Realistic modeling and rendering of plant ecosystems. In *Proceedings of SIGGRAPH '98*, Computer Graphics Proceedings, Annual Conference Series, 275–86.
- Devlin, K., A. Chalmers, A. Wilkie, and W. Purgathofer. 2002. Tone reproduction and physically based spectral rendering. In D. Fellner and R. Scopigno (Eds.), *Proceedings of Eurographics 2002*, 101–23. The Eurographics Association.
- Dick, J., and F. Pillichshammer. 2010. *Digital Nets and Sequences: Discrepancy Theory and Quasi-Monte Carlo Integration*. Cambridge: Cambridge University Press.
- Dippé, M. A. Z., and E. H. Wold. 1985. Antialiasing through stochastic sampling. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19, 69–78.
- Dobkin, D. P., and D. P. Mitchell. 1993. Random-edge discrepancy of supersampling patterns. In *Proceedings of Graphics Interface 1993*, Toronto, Ontario, 62–69. Canadian Information Processing Society.
- Dobkin, D. P., D. Eppstein, and D. P. Mitchell. 1996. Computing the discrepancy with applications to supersampling patterns. *ACM Transactions on Graphics* 15(4), 354–76.
- Donikian, M., B. Walter, K. Bala, S. Fernandez, and D. P. Greenberg. 2006. Accurate direct illumination using iterative adaptive sampling. *IEEE Transactions on Visualization and Computer Graphics* 12(3), 353–64.
- Donnelly, W. 2005. Per-pixel displacement mapping with distance functions. In M. Pharr (Ed.), *GPU Gems 2*. Reading, Massachusetts: Addison-Wesley.
- Donner, C. 2006. Towards realistic image synthesis of scattering materials. Ph.D. thesis, University of California, San Diego.
- Donner, C., and H. W. Jensen. 2005. Light diffusion in multi-layered translucent materials. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)* 24(3), 1032–39.
- Donner, C., and H. W. Jensen. 2006. A spectral BSSRDF for shading human skin. *Rendering Techniques 2006: 17th Eurographics Workshop on Rendering*, 409–17.
- Donner, C., T. Weyrich, E. d'Eon, R. Ramamoorthi, and S. Rusinkiewicz. 2008. A layered, heterogeneous reflectance model for acquiring and rendering human skin. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2008)* 27(5), 140:1–140:12.
- Donner, C., J. Lawrence, R. Ramamoorthi, T. Hachisuka, H. W. Jensen, and S. Nayar. 2009. An empirical BSSRDF model. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2009)* 28(3), 30:1–30:10.
- Doo, D., and M. Sabin. 1978. Behaviour of recursive division surfaces near extraordinary points. *Computer-Aided Design* 10, 356–60.
- Dorsey, J. O., F. X. Sillion, and D. P. Greenberg. 1991. Design and simulation of opera lighting and projection effects. In *Computer Graphics (Proceedings of SIGGRAPH '91)*, 25, 41–50.
- Dorsey, J., and P. Hanrahan. 1996. Modeling and rendering of metallic patinas. In *Proceedings of SIGGRAPH '96*, 387–96.
- Dorsey, J., H. K. Pedersen, and P. M. Hanrahan. 1996. Flow and changes in appearance. In *Proceedings of SIGGRAPH '96*, Computer Graphics Proceedings, Annual Conference Series, 411–20.
- Dorsey, J., A. Edelman, J. Legakis, H. W. Jensen, and H. K. Pedersen. 1999. Modeling and rendering of weathered stone. In *Proceedings of SIGGRAPH '99*, Computer Graphics Proceedings, Annual Conference Series, 225–34.

- Doyle, M. J., C. Fowler, and M. Manzke. 2013. A hardware unit for fast SAH-optimised BVH construction. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2013)* 32(4), 139:1–139:10.
- Drebin, R. A., L. Carpenter, and P. Hanrahan. 1988. Volume rendering. *Computer Graphics (Proceedings of SIGGRAPH '88)*, 22, 65–74.
- Drepper, U. 2007. What every programmer should know about memory. people.redhat.com/drepper/cpumemory.pdf.
- Drew, M., and G. Finlayson. 2003. Multispectral rendering without spectra. *Journal of the Optical Society of America A* 20(7), 1181–93.
- Driemeyer, T., and R. Herken. 2002. *Programming mental ray*. Wien: Springer-Verlag.
- Du, S.-P., S.-M. Hu, and R. R. Martin. 2013. Semiregular solid texturing from 2D image exemplars. *IEEE Transactions on Visualization and Computer Graphics* 19(3), 460–69.
- Duff, T. 1985. Compositing 3-D rendered images. *Computer Graphics (Proceedings of SIGGRAPH '85)*, 19, 41–44.
- Dunbar, D., and G. Humphreys. 2006. A spatial data structure for fast Poisson-disk sample generation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006)* 25(3), 503–08.
- Dungan, W. Jr., A. Stenger, and G. Sutty. 1978. Texture tile considerations for raster graphics. *Computer Graphics (Proceedings of SIGGRAPH '78)*, 12, 130–34.
- Dupuy, J., E. Heitz, J.-C. Iehl, P. Poulin, F. Neyret, and V. Ostromoukhov. 2013. Linear efficient antialiased displacement and reflectance mapping. *ACM Transactions on Graphics* 32(6).
- Dupuy, J., E. Heitz, J.-C. Iehl, P. Poulin, and V. Ostromoukhov. 2015. Extracting microfacet-based BRDF parameters from arbitrary materials with power iterations. *Computer Graphics Forum (Proceedings of the 2015 Eurographics Symposium on Rendering)* 34(4), 21–30.
- Durand, F., N. Holzschuch, C. Soler, E. Chan, and F. X. Sillion. 2005. A frequency analysis of light transport. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)* 24(3), 1115–26.
- Dutr  , P. 2003. Global illumination compendium. www.cs.kuleuven.ac.be/~phil/GI/.
- Ebeida, M., A. Davidson, A. Patney, P. Knupp, S. Mitchell, and J. D. Owens. 2011. Efficient maximal Poisson-disk sampling. *ACM Transactions on Graphics* 30(4), 49:1–49:12.
- Ebeida, M., S. Mitchell, A. Patney, A. Davidson, and J. D. Owens. 2012. A simple algorithm for maximal Poisson-disk sampling in high dimensions. *Computer Graphics Forum (Proceedings of Eurographics 2012)* 31(2), 785–94.
- Eberly, D. H. 2001. *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*. San Francisco: Morgan Kaufmann.
- Eberly, D. 2011. A fast and accurate algorithm for computing SLERP. *Journal of Graphics, GPU, and Game Tools* 15(3), 161–76.
- Ebert, D., F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley. 2003. *Texturing and Modeling: A Procedural Approach*. San Francisco: Morgan Kaufmann.
- Edwards, D., S. Boulos, J. Johnson, P. Shirley, M. Ashikhmin, M. Stark, and C. Wyman. 2005. The halfway vector disk for BRDF modeling. *ACM Transactions on Graphics* 25(1), 1–18.
- Egan, K., Y.-T. Tseng, N. Holzschuch, F. Durand, and R. Ramamoorthi. 2009. Frequency analysis and sheared reconstruction for rendering motion blur. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2009)* 28(3), 93:1–93:13.

- Eisemann, M., M. Magnor, T. Grosch, and S. Müller. 2007. Fast ray/axis-aligned bounding box overlap tests using ray slopes. *Journal of Graphics, GPU, and Game Tools* 12(4), 35–46.
- Eisenacher, C., G. Nichols, A. Selle, and B. Burley. 2013. Sorted deferred shading for production path tracing. *Computer Graphics Forum (Proceedings of the 2013 Eurographics Symposium on Rendering)* 32(4), 125–32.
- Eldar, Y. C., and T. Michaeli. 2009. Beyond bandlimited sampling. *IEEE Signal Processing Magazine* 26(3), 48–68.
- Elek, O., P. Bauszat, T. Ritschel, M. Magnor, and H.-P. Seidel. 2014. Spectral ray differentials. *Computer Graphics Forum (Proceedings of the 2014 Eurographics Symposium on Rendering)* 33(4), 113–22.
- Ericson, C. 2004. *Real-Time Collision Detection*. Morgan Kaufmann Series in Interactive 3D Technology. San Francisco: Morgan Kaufmann.
- Ernst, M., and G. Greiner. 2007. Early split clipping for bounding volume hierarchies. *IEEE Symposium on Interactive Ray Tracing*, 73–78.
- Ernst, M., and G. Greiner. 2008. Multi bounding volume hierarchies. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing 2008*, 35–40.
- Evans, G., and M. McCool. 1999. Stratified wavelength clusters for efficient spectral Monte Carlo rendering. In *Proceedings of Graphics Interface 1999*, 42–49.
- Fabianowski, B., C. Fowler, and J. Dingliana. 2009. A cost metric for scene-interior ray origins. In *Short Paper Proceedings of the 30th Annual Conference of the European Association for Computer Graphics (Eurographics 2009)*, 49–50.
- Fan, S., S. Chenney, and Y.-C. Lai. 2005. Metropolis photon sampling with optional user guidance. In *Rendering Techniques 2005: 16th Eurographics Workshop on Rendering*, 127–38.
- Fante, R. L. 1981. Relationship between radiative-transport theory and Maxwell's equations in dielectric media. *Journal of the Optical Society of America* 71(4), 460–468.
- Farin, G. 2001. *Curves and Surfaces for CAGD: A Practical guide*, (5th ed.). San Francisco: Morgan Kaufmann.
- Farmer, D. F. 1981. Comparing the 4341 and M80/40. *Computerworld* 15(6).
- Farrell, T., M. Patterson, and B. Wilson. 1992. A diffusion theory model of spatially resolved, steady-state diffuse reflectance for the noninvasive determination of tissue optical properties *in vivo*. *Med. Phys.* 19(4), 879–88.
- Fatahalian, K. 2008. Running code at a teraflop: how GPU shader cores work. In *Beyond Programmable Shading, SIGGRAPH 2008 Course Notes*.
- Fattal, R. 2009. Participating media illumination using light propagation maps. *ACM Transactions on Graphics* 28(1), 7:1–7:11.
- Faure, H. 1992. Good permutations for extreme discrepancy. *Journal of Number Theory* 42, 47–56.
- Fedkiw, R., J. Stam, and H. W. Jensen. 2001. Visual simulation of smoke. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, 15–22.
- Feibusch, E. A., M. Levoy, and R. L. Cook. 1980. Synthetic texturing using digital filters. *Computer Graphics (Proceedings of SIGGRAPH '80)*, 14, 294–301.
- Fernandez, S., K. Bala, and D. P. Greenberg. 2002. Local illumination environments for direct lighting acceleration. In *Rendering Techniques 2002: 13th Eurographics Workshop on Rendering*, 7–14.

- Ferwerda, J. A. 2001. Elements of early vision for computer graphics. *IEEE Computer Graphics and Applications* 21(5), 22–33.
- Fisher, M., K. Fatahalian, S. Boulos, K. Akeley, W. R. Mark, and P. Hanrahan. 2009. DiagSplit: parallel, crack-free, adaptive tessellation for micropolygon rendering. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2009)* 28(5), 150:1–150:10.
- Fishman, G. S. 1996. *Monte Carlo: Concepts, Algorithms, and Applications*. New York: Springer-Verlag.
- Fleischer, K., D. Laidlaw, B. Currin, and A. H. Barr. 1995. Cellular texture generation. In *Proceedings of SIGGRAPH '95*, Computer Graphics Proceedings, Annual Conference Series, 239–48.
- Foley, T., and J. Sugerman. 2005. KD-tree acceleration structures for a GPU raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 15–22.
- Fournier, A. 1992. Normal distribution functions and multiple surfaces. In *Graphics Interface '92 Workshop on Local Illumination*, 45–52.
- Fournier, A., and E. Fiume. 1988. Constant-time filtering with space-variant kernels. *Computer Graphics (SIGGRAPH '88 Proceedings)*, 22, 229–38.
- Fournier, A., D. Fussel, and L. Carpenter. 1982. Computer rendering of stochastic models. *Communications of the ACM* 25(6), 371–84.
- Fraser, C., and D. Hanson. 1995. *A Retargetable C Compiler: Design and Implementation*. Reading, Massachusetts: Addison-Wesley.
- Friedel, I., and A. Keller. 2000. Fast generation of randomized low discrepancy point sets. In *Monte Carlo and Quasi-Monte Carlo Methods 2000*, 257–73. Berlin: Springer-Verlag.
- Frisvad, J., N. Christensen, and H. W. Jensen. 2007. Computing the scattering properties of participating media using Lorenz-Mie theory. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007)* 26(3), 60:1–60:10.
- Frisvad, J. R., T. Hachisuka, and T. K. Kjeldsen. 2014. Directional dipole model for subsurface scattering. *ACM Transactions on Graphics* 34(1), 5:1–5:12.
- Fuchs, C., T. Chen, M. Goesele, H. Theisel, and H.-P. Seidel. 2007. Density estimation for dynamic volumes. *Computers and Graphics* 31(2), 205–11.
- Fujimoto, A., T. Tanaka, and K. Iwata. 1986. Arts: accelerated ray-tracing system. *IEEE Computer Graphics and Applications* 6(4), 16–26.
- Galerne, B., A. Lagae, S. Lefebvre, and G. Drettakis. 2012. Gabor noise by example. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2012)* 31(4), 73:1–73:9.
- Garanzha, K. 2009. The use of precomputed triangle clusters for accelerated ray tracing in dynamic scenes. *Computer Graphics Forum (Proceedings of the 2009 Eurographics Symposium on Rendering)* 28(4), 1199–1206.
- García, R., C. Ureña, and M. Sbert. 2012. Description and solution of an unreported intrinsic bias in photon mapping density estimation with constant kernel. *Computer Graphics Forum* 31(1), 33–41.
- Gardner, G. Y. 1984. Simulation of natural scenes using textured quadric surfaces. *Computer Graphics (SIGGRAPH '84 Proceedings)*, 18, 11–20.
- Gardner, G. Y. 1985. Visual simulation of clouds. *Computer Graphics (Proceedings of SIGGRAPH '85)*, 19, 297–303.

- Gardner, R. P., H. K. Choi, M. Mickael, A. M. Yacout, Y. Yin, and K. Verghese. 1987. Algorithms for forcing scattered radiation to spherical, planar circular, and right circular cylindrical detectors for Monte Carlo simulation. *Nuclear Science and Engineering* 95, 245–56.
- Garanzha, K., J. Pantaleoni, D. McAllister. 2011. Simpler and faster HLBVH with work queues. In *Proceedings of High Performance Graphics 2011*, 59–64.
- Gershbein, R., and P. M. Hanrahan. 2000. A fast relighting engine for interactive cinematic lighting design. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, 353–58.
- Gershun, A. 1939. The light field. *Journal of Mathematics and Physics* 18, 51–151.
- Georgiev, I., J. Křivánek, T. Davidovič, and P. Slusallek. 2012. Light transport simulation with vertex connection and merging. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2012)* 31(6), 192:1–192:10.
- Georgiev, I., J. Křivánek, T. Hachisuka, D. Nowrouzezahrai, and W. Jarosz. 2013. Joint importance sampling of low-order volumetric scattering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2013)* 32(6), 164:1–164:14.
- Georgiev, I., and P. Slusallek. 2008. RTfact: generic concepts for flexible and high performance ray tracing. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, 115–22.
- Ghosh, A., and W. Heidrich. 2006. Correlated visibility sampling for direct illumination. *The Visual Computer* 22(9–10), 693–701.
- Ghosh, A., T. Hawkins, P. Peers, S. Frederiksen, and P. Debevec. 2008. Practical modeling and acquisition of layered facial reflectance. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2008)* 27(5), 139:1–139:10.
- Gijsenij, A., T. Gevers, J. van de Weijer. 2011. Computational color constancy: survey and experiments. *IEEE Transactions on Image Processing* 20(9), 2475–89.
- Gilet, G., B. Sauvage, K. Vanhoey, J.-M. Dischler, and D. Ghazanfarpour. 2014. Local random-phase noise for procedural texturing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2014)* 33(6), 195:1–195:11.
- Gkioulekas, I., S. Zhao, K. Bala, T. Zickler, and A. Levin. 2013a. Inverse volume rendering with material dictionaries. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2013)* 32(6), 162:1–162:13.
- Gkioulekas, I., B. Xiao, S. Zhao, E. H. Adelson, T. Zickler, and K. Bala. 2013b. Understanding the role of phase function in translucent appearance. *ACM Transactions on Graphics* 32(5), 147:1–147:19.
- Glassner, A. 1984. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications* 4(10), 15–22.
- Glassner, A. 1988. Spacetime ray tracing for animation. *IEEE Computer Graphics & Applications* 8(2), 60–70.
- Glassner, A. (Ed.) 1989a. *An Introduction to Ray Tracing*. San Diego: Academic Press.
- Glassner, A. 1989b. How to derive a spectrum from an RGB triplet. *IEEE Computer Graphics and Applications* 9(4), 95–99.
- Glassner, A. 1993. Spectrum: an architecture for image synthesis, research, education, and practice. *Developing Large-Scale Graphics Software Toolkits, SIGGRAPH '93 Course Notes*, 3, 1-14–1-43.
- Glassner, A. 1994. A model for fluorescence and phosphorescence. In *Proceedings of the Fifth Eurographics Workshop on Rendering*, 57–68.

- Glassner, A. 1995. *Principles of Digital Image Synthesis*. San Francisco: Morgan Kaufmann.
- Glassner, A. 1999. An open and shut case. *IEEE Computer Graphics and Applications* 19(3), 82–92.
- Goesele, M., X. Granier, W. Heidrich, and H.-P. Seidel. 2003. Accurate light source acquisition and rendering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003)* 22(3), 621–30.
- Goesele, M., H. Lensch, J. Lang, C. Fuchs, and H.-P. Seidel. 2004. DISCO—Acquisition of translucent objects. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004)* 23(3), 844–53.
- Goldberg, A., M. Zwicker, and F. Durand. 2008. Anisotropic noise. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008)* 27(3), 54:1–54:8.
- Goldberg, D. 1991. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys* 23(1), 5–48.
- Goldman, D. B. 1997. Fake fur rendering. In *Proceedings of SIGGRAPH '97*, Computer Graphics Proceedings, Annual Conference Series, 127–34.
- Goldman, R. 1985. Illicit expressions in vector algebra. *ACM Transactions on Graphics* 4(3), 223–43.
- Goldsmith, J., and J. Salmon. 1987. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications* 7(5), 14–20.
- Goldstein, R. A., and R. Nagel. 1971. 3-D visual simulation. *Simulation* 16(1), 25–31.
- Goral, C. M., K. E. Torrance, D. P. Greenberg, and B. Battaile. 1984. Modeling the interaction of light between diffuse surfaces. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '84)*, 213–22.
- Gortler, S. J., R. Grzeszczuk, R. Szeliski, and M. F. Cohen. 1996. The lumigraph. In *Proceedings of SIGGRAPH '96*, Computer Graphics Proceedings, Annual Conference Series, 43–54.
- Gray, A. 1993. *Modern Differential Geometry of Curves and Surfaces*. Boca Raton, Florida: CRC Press.
- Green, S. A., and D. J. Paddon. 1989. Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics and Applications* 9(6), 12–26.
- Greenberg, D. P., K. E. Torrance, P. S. Shirley, J. R. Arvo, J. A. Ferwerda, S. Pattanaik, E. P. F. Lafontaine, B. Walter, S.-C. Foo, and B. Trumbore. 1997. A framework for realistic image synthesis. In *Proceedings of SIGGRAPH '97*, Computer Graphics Proceedings, Annual Conference Series, 477–94.
- Greene, N. 1986. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications* 6(11), 21–29.
- Greene, N., and P. S. Heckbert. 1986. Creating raster Omnimax images from multiple perspective views using the elliptical weighted average filter. *IEEE Computer Graphics and Applications* 6(6), 21–27.
- Gribble, C., and K. Ramani. 2008. Coherent ray tracing via stream filtering. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, 59–66.
- Gritz, L., and E. d'Eon. 2008. The importance of being linear. In H. Nguyen (Ed.), *GPU Gems 3*. Boston, Massachusetts: Addison-Wesley.
- Gritz, L., and J. K. Hahn. 1996. BMRT: a global illumination implementation of the RenderMan standard. *Journal of Graphics Tools* 1(3), 29–47.

- Gritz, L., C. Stein, C. Kulla, and A. Conty. 2010. Open Shading Language. *SIGGRAPH 2010 Talks*.
- Grosjean, C. C. 1956. A high accuracy approximation for solving multiple scattering problems in infinite homogeneous media *Nuovo Cimento* 3(6), 1262–75.
- Grünschloß, L., J. Hanika, R. Schwede, and A. Keller. 2008. (t, m, s)-nets and maximized minimum distance. In A. Keller, S. Heinrich, and H. Niederreiter (eds.), *Monte Carlo and Quasi-Monte Carlo Methods 2006*. Berlin: Springer Verlag.
- Grünschloß, L., and A. Keller. 2009. (t, m, s)-nets and maximized minimum distance, Part II. In P. L'Ecuyer and A. Owen (Eds.), *Monte Carlo and Quasi-Monte Carlo Methods 2008*.
- Grünschloß, L., M. Raab, and A. Keller. 2012. Enumerating quasi-Monte Carlo point sequences in elementary intervals. In H. Wozniakowski and L. Plaskota (Eds.), *Monte Carlo and Quasi-Monte Carlo Methods 2010*.
- Grünschloß, L., M. Stich, S. Nawaz, and A. Keller. 2011. MSBVH: an efficient acceleration data structure for ray traced motion blur. In *Proceedings of High Performance Graphics 2011*, 65–70.
- Grunwald, D., B. G. Zorn, and R. Henderson. 1993. Improving the cache locality of memory allocation. In *SIGPLAN Conference on Programming Language Design and Implementation*, 177–86.
- Gu, J., S. K. Nayar, E. Grinspun, P. N. Belhumeur, and R. Ramamoorthi. 2013a. Compressive structured light for recovering inhomogeneous participating media. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35(3).
- Gu, Y., Y. He, K. Fatahalian, and G. Blelloch. 2013b. Efficient BVH construction via approximate agglomerative clustering. *Proceesings of High Performance Graphics 2013*, 81–88.
- Guertin, J.-P., M. McGuire, and D. Nowrouzezahrai. 2014. A fast and stable feature-aware motion blur filter. In *Proceedings of High Performance Graphics 2014*.
- Günther, J., T. Chen, M. Goesele, I. Wald, and H.-P. Seidel. 2005. Efficient acquisition and realistic rendering of car paint. In *Proceedings of Vision, Modeling, and Visualization (VMV)*, 487–94.
- Günther, J., S. Popov, H. P. Seidel, and P. Slusallek. 2007. Realtime ray tracing on GPU with BVH-based packet traversal. In *IEEE Symposium on Interactive Ray Tracing*, 113–18.
- Guthe, S., and P. Heckbert. 2005. Non-power-of-two mipmapping. *NVIDIA Technical Report*, developer.nvidia.com/object/np2_mipmapping.html.
- Habel, R., P. H. Christensen, and W. Jarosz. 2013. Photon beam diffusion: a hybrid Monte Carlo method for subsurface scattering. *Computer Graphics Forum (Proceedings of the 2013 Eurographics Symposium on Rendering)* 32(4), 27–37.
- Haber, J., M. Magnor, and H.-P. Seidel. 2005a. Physically-based simulation of twilight phenomena. *ACM Transactions on Graphics* 24(4), 1353–73.
- Haber, T., T. Mertens, P. Bekaert, and F. Van Reeth. 2005b. A computational approach to simulate subsurface light diffusion in arbitrarily shaped objects. In *Proceedings of Graphics Interface 2005*, 79–86.
- Hachisuka, T. 2005. High-quality global illumination rendering using rasterization. In M. Pharr (Ed.), *GPU Gems 2: Programming Techniques for High Performance Graphics and General-Purpose Computation*. Reading, Massachusetts: Addison-Wesley.
- Hachisuka, T. 2011. Robust light transport simulation using progressive density estimation. Ph.D. thesis, University of California, San Diego.

- Hachisuka, T., W. Jarosz, and H. W. Jensen. 2010. A progressive error estimation framework for photon density estimation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2010)* 29(6), 144:1–144:12.
- Hachisuka, T., W. Jarosz, R. P. Weistroffer, K. Dale, G. Humphreys, M. Zwicker, and H. W. Jensen. 2008a. Multidimensional adaptive sampling and reconstruction for ray tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008)* 27(3), 33:1–33:10.
- Hachisuka, T., and H. W. Jensen. 2009. Stochastic progressive photon mapping. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2009)* 28(5), 141:1–141:8.
- Hachisuka, T., and H. W. Jensen. 2011. Robust adaptive photon tracing using photon path visibility. *ACM Transactions on Graphics* 30(5), 114:1–114:11.
- Hachisuka, T., A. S. Kaplanyan, and C. Dachsbacher. 2014. Multiplexed Metropolis light transport. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014)* 33(4), 100:1–100:10.
- Hachisuka, T., S. Ogaki, and H. W. Jensen. 2008b. Progressive photon mapping. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2008)* 27(5), 130:1–130:8.
- Hachisuka, T., J. Pantaleoni, and H. W. Jensen. 2012. A path space extension for robust light transport simulation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2012)* 31(6), 191:1–191:10.
- Haines, E. A. 1989. Essential ray tracing algorithms. In A. Glassner (Ed.), *An Introduction to Ray Tracing*, 33–78. San Diego: Academic Press.
- Haines, E. A. 1994. Point in polygon strategies. In P. Heckbert (Ed.), *Graphics Gems IV*, 24–46. San Diego: Academic Press.
- Haines, E. A., and D. P. Greenberg. 1986. The light buffer: a shadow testing accelerator. *IEEE Computer Graphics and Applications* 6(9), 6–16.
- Haines, E. A., and J. R. Wallace. 1994. Shaft culling for efficient ray-traced radiosity. In *Second Eurographics Workshop on Rendering (Photorealistic Rendering in Computer Graphics)*. Also in *SIGGRAPH 1991 Frontiers in Rendering Course Notes*.
- Hakura, Z. S., and A. Gupta. 1997. The design and analysis of a cache architecture for texture mapping. In *Proceedings of the 24th International Symposium on Computer Architecture*, Denver, Colorado, 108–20.
- Hall, R. 1989. *Illumination and Color in Computer Generated Imagery*. New York: Springer-Verlag.
- Hall, R. 1999. Comparing spectral color computation methods. *IEEE Computer Graphics and Applications* 19(4), 36–46.
- Hall, R. A., and D. P. Greenberg. 1983. A testbed for realistic image synthesis. *IEEE Computer Graphics and Applications* 3(8), 10–20.
- Hammersley, J., and D. Handscomb. 1964. *Monte Carlo Methods*. New York: John Wiley.
- Han, C., B. Sun, R. Ramamoorthi, and E. Grinspun. 2007. Frequency domain normal map filtering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007)* 26(3), 28:1–28:11.
- Hanika, J., and C. Dachsbaucher. 2014. Efficient Monte Carlo rendering with realistic lenses. *Computer Graphics Forum (Proceedings of Eurographics 2014)* 33(2), 323–32.
- Hanika, J., M. Droske, and L. Fascione. 2015a. Manifold next event estimation. *Computer Graphics Forum (Proceedings of the 2015 Eurographics Symposium on Rendering)* 34(4), 87–97.

- Hanika, J., A. Kaplanyan, and C. Dachsbacher. 2015b. Improved half vector space light transport. *Computer Graphics Forum (Proceedings of the 2015 Eurographics Symposium on Rendering)* 34(4), 65–74.
- Hanika, J., A. Keller, and H. P. A. Lensch. 2010. Two-level ray tracing with reordering for highly complex scenes. In *Proceedings of Graphics Interface 2010*, 145–52.
- Hanrahan, P. 1983. Ray tracing algebraic surfaces. *Computer Graphics (Proceedings of SIGGRAPH '83)*, 17, 83–90.
- Hanrahan, P., and W. Krueger. 1993. Reflection from layered surfaces due to subsurface scattering. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, 165–74.
- Hanrahan, P., and J. Lawson. 1990. A language for shading and lighting calculations. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24, 289–98.
- Hansen, J. E., and L. D. Travis. 1974. Light scattering in planetary atmospheres. *Space Science Reviews* 16, 527–610.
- Hart, D., P. Dutré, and D. P. Greenberg. 1999. Direct illumination with lazy visibility evaluation. In *Proceedings of SIGGRAPH '99*, Computer Graphics Proceedings, Annual Conference Series, 147–54.
- Hart, J. C. 1996. Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer* 12(9), 527–45.
- Hart, J. C., D. J. Sandin, and L. H. Kauffman. 1989. Ray tracing deterministic 3-D fractals. *Computer Graphics (Proceedings of SIGGRAPH '89)*, 23, 289–96.
- Hasinoff, S. W., and K. N. Kutulakos. 2011. Light-efficient photography. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33(11), 2203–14.
- Havran, V. 2000. Heuristic ray shooting algorithms. Ph.D. thesis, Czech Technical University.
- Havran, V., and J. Bittner. 2002. On improving kd-trees for ray shooting. In *Proceedings of WSCG 2002 Conference*, 209–17.
- Havran, V., R. Herzog, and H.-P. Seidel. 2005. Fast final gathering via reverse photon mapping. *Computer Graphics Forum (Proceedings of Eurographics 2005)* 24(3), 323–34.
- Havran, V., R. Herzog, and H.-P. Seidel. 2006. On the fast construction of spatial hierarchies for ray tracing. In *IEEE Symposium on Interactive Ray Tracing*, 71–80.
- Hawkins, T., P. Einarsson, and P. Debevec. 2005. Acquisition of time-varying participating media. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)* 24(3), 812–15.
- Hecht, E. 2002. *Optics*. Reading, Massachusetts: Addison-Wesley.
- Heckbert, P. S. 1984. The Mathematics of Quadric Surface Rendering and SOID. *3-D Technical Memo*, New York Institute of Technology Computer Graphics Lab.
- Heckbert, P. S. 1986. Survey of texture mapping. *IEEE Computer Graphics and Applications* 6(11), 56–67.
- Heckbert, P. S. 1987. Ray tracing JELL-O brand gelatin. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4), 73–74.
- Heckbert, P. S. 1989a. Image zooming source code. www-2.cs.cmu.edu/~ph/src/zoom/.
- Heckbert, P. S. 1989b. Fundamentals of texture mapping and image warping. M.S. thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley.
- Heckbert, P. S. 1990a. What are the coordinates of a pixel? In A. S. Glassner (Ed.), *Graphics Gems I*, 246–48. San Diego: Academic Press.

- Heckbert, P. S. 1990b. Adaptive radiosity textures for bidirectional ray tracing. *Computer Graphics (Proceedings of SIGGRAPH '90)*, 24, 145–54.
- Heckbert, P. S., and P. Hanrahan. 1984. Beam tracing polygonal objects. In *Computer Graphics (Proceedings of SIGGRAPH '84)*, 18, 119–27.
- Heidrich, W., and H.-P. Seidel. 1998. Ray-tracing procedural displacement shaders. In *Proceedings of Graphics Interface 1998*, 8–16.
- Heidrich, W., J. Kautz, P. Slusallek, and H.-P. Seidel. 1998. Canned lightsources. In *Rendering Techniques '98: Proceedings of the Eurographics Rendering Workshop*, 293–300.
- Heitz, E. 2014a. Understanding the masking-shadowing function in microfacet-based BRDFs. *Journal of Computer Graphics Techniques (JCGT)* 3(2), 32–91.
- Heitz, E. 2015. Derivation of the microfacet $\Lambda(\omega)$ function. Personal communication.
- Heitz, E., C. Bourlier, and N. Pinel. 2013. Correlation effect between transmitter and receiver azimuthal directions on the illumination function from a random rough surface. *Waves in Random and Complex Media* 23(3), 318–35.
- Heitz, E., and E. d'Eon. 2014. Importance sampling microfacet-based BSDFs using the distribution of visible normals. *Computer Graphics Forum (Proceedings of The 2014 Eurographics Symposium on Rendering)* 33(4), 103–12.
- Heitz, E., J. Dupuy, C. Crassin, and C. Dachsbacher. 2015. The SGGX microflake distribution. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2015)* 34(4), 48:1–48:11.
- Heitz, E., and F. Neyret. 2012. Representing appearance and pre-filtering subpixel data in sparse voxel octrees. In *Proceedings of High Performance Graphics 2012*, 125–34.
- Heitz, E., D. Nowrouzezahrai, P. Poulin, and F. Neyret. 2014. Filtering non-linear transfer functions on surfaces. *IEEE Transactions on Visualization and Computer Graphics* 20(7), 996–1008.
- Heney, L. G., and J. L. Greenstein. 1941. Diffuse radiation in the galaxy. *Astrophysical Journal* 93, 70–83.
- Hery, C. 2003. Implementing a skin BSSRDF. *SIGGRAPH 2003 RenderMan Course Notes*.
- Hery, C., M. Kass, and J. Ling. 2014. Geometry into shading. *Pixar Technical Memo 14-04*.
- Hery, R., and R. Ramamoorthi. 2012. Importance sampling of reflection from hair fibers. *Journal of Computer Graphics Techniques (JCGT)* 1(1), 1–17.
- Herzog, R., V. Havran, S. Kinuwaki, K. Myszkowski, and H.-P. Seidel. 2007. Global illumination using photon ray splatting. *Computer Graphics Forum (Proceedings of Eurographics 2007)* 26(3), 503–13.
- Higham, N. 1986. Computing the polar decomposition—with applications. *SIAM Journal of Scientific and Statistical Computing* 7(4), 1160–74.
- Higham, N. J. 2002. *Accuracy and Stability of Numerical Algorithms* (2nd ed.). Philadelphia: Society for Industrial and Applied Mathematics.
- Hoberock, J. 2008. Accelerating physically-based light transport algorithms. Ph.D. thesis, University of Illinois at Urbana-Champaign.
- Hoberock, J., V. Lu, Y. Jia, J. Hart. 2009. Stream compaction for deferred shading. In *Proceedings of High Performance Graphics 2009*, 173–80.
- Hoffmann, C. M. 1989. *Geometric and Solid Modeling: An Introduction*. San Francisco: Morgan Kaufmann.

- Hoppe, H., T. DeRose, T. Duchamp, M. Halstead, H. Jin, J. McDonald, J. Schweitzer, and W. Stuetzle. 1994. Piecewise smooth surface reconstruction. In *Proceedings of SIGGRAPH '94*, Computer Graphics Proceedings, Annual Conference Series, Orlando, Florida, 295–302.
- Hošek, L., and A. Wilkie. 2012. An analytic model for full spectral sky-dome radiance. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2012)* 31(4), 95:1–95:9.
- Hošek, L., and A. Wilkie. 2013. Adding a solar-radiance function to the Hošek–Wilkie skylight model. *IEEE Computer Graphics and Applications* 33(3), 44–52.
- Hullin, M. B., J. Hanika., and W. Heidrich. 2012. Polynomial optics: a construction kit for efficient ray-tracing of lens systems. *Computer Graphics Forum (Proceedings of the 2012 Eurographics Symposium on Rendering)* 31(4), 1375–83.
- Hunt, W. 2008. Corrections to the surface area metric with respect to mail-boxing. In *IEEE Symposium on Interactive Ray Tracing*, 77–80.
- Hunt, W., and B. Mark. 2008a. Ray-specialized acceleration structures for ray tracing. In *IEEE Symposium on Interactive Ray Tracing*, 3–10.
- Hunt, W., and B. Mark. 2008b. Adaptive acceleration structures in perspective space. In *IEEE Symposium on Interactive Ray Tracing*, 117–17.
- Hunt, W., W. Mark, and G. Stoll. 2006. Fast kd-tree construction with an adaptive error-bounded heuristic. In *IEEE Symposium on Interactive Ray Tracing*, 81–88.
- Hurley, J., A. Kapustin, A. Reshetov, and A. Soupikov. 2002. Fast ray tracing for modern general purpose CPU. In *Proceedings of GraphiCon 2002*.
- Igarashi, T., K. Nishino, and S. K. Nayar. 2007. The appearance of human skin: a survey. *Foundations and Trends in Computer Graphics and Vision* 3(1), 1–95.
- Igehy, H. 1999. Tracing ray differentials. In *Proceedings of SIGGRAPH '99*, Computer Graphics Proceedings, Annual Conference Series, 179–86.
- Igehy, H., M. Eldridge, and K. Proudfoot. 1998. Prefetching in a texture cache architecture. In *1998 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 133–42.
- Igehy, H., M. Eldridge, and P. Hanrahan. 1999. Parallel texture caching. In *1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 95–106.
- Illuminating Engineering Society of North America. 2002. IESNA standard file format for electronic transfer of photometric data. BSR/IESNA Publication LM-63-2002. www.iesna.org.
- Immel, D. S., M. F. Cohen, and D. P. Greenberg. 1986. A radiosity method for non-diffuse environments. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, Volume 20, 133–42.
- Institute of Electrical and Electronic Engineers. 1985. IEEE standard 754-1985 for binary floating-point arithmetic. Reprinted in *SIGPLAN* 22(2), 9–25.
- Institute of Electrical and Electronic Engineers. 2008. IEEE standard 754-2008 for binary floating-point arithmetic.
- International Electrotechnical Commission (IEC). 1999. Multimedia systems and equipment—Colour measurement and management—Part 2-1: Colour management—Default RGB colour space—sRGB. IEC Standard 61966-2-1.
- Irawan, P. 2008. Appearance of woven cloth. Ph.D. thesis, Cornell University.
- Irawan, P., and S. Marschner. 2012. Specular reflection from woven cloth. *ACM Transactions on Graphics* 31(1).
- Ishimaru, A. 1978. *Wave Propagation and Scattering in Random Media*. Oxford: Oxford University Press.

- Ize, T. 2013. Robust BVH ray traversal. *Journal of Computer Graphics Techniques (JCGT)* 2(2), 12–27.
- Ize, T., and C. Hansen. 2011. RTSAH traversal order for occlusion rays. *Computer Graphics Forum (Proceedings of Eurographics 2011)* 30(2), 295–305.
- Ize, T., P. Shirley, and S. Parker. 2007. Grid creation strategies for efficient ray tracing. In *IEEE Symposium on Interactive Ray Tracing*, 27–32.
- Ize, T., I. Wald, and S. Parker. 2008. Ray tracing with the BSP tree. In *IEEE Symposium on Interactive Ray Tracing*, 159–66.
- Ize, T., I. Wald, C. Robertson, and S. G. Parker. 2006. An evaluation of parallel grid construction for ray tracing dynamic scenes. *IEEE Symposium on Interactive Ray Tracing*, 47–55.
- Jackson, W. H. 1910. The solution of an integral equation occurring in the theory of radiation. *Bulletin of the American Mathematical Society* 16, 473–75.
- Jacobs, D. E., J. Baek, and M. Levoy. 2012. Focal stack compositing for depth of field control. *Stanford Computer Graphics Laboratory Technical Report*, CSTR 2012-1.
- Jakob, W. 2010. Mitsuba renderer. <http://www.mitsuba-renderer.org>.
- Jakob, W. 2013. Light transport on path-space manifolds. Ph.D. thesis, Cornell University.
- Jakob, W., A. Arbree, J. T. Moon, K. Bala, and M. Steve. 2010. A radiative transfer framework for rendering materials with anisotropic structure. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010)* 29(4), 53:1–53:13.
- Jakob, W., E. d'Eon, O. Jakob, and S. Marschner. 2014a. A comprehensive framework for rendering layered materials. *ACM Transactions on Graphics* 33(4), 118:1–118:14.
- Jakob, W., M. Hašan, L.-Q. Yan, J. Lawrence, R. Ramamoorthi, and S. Marschner. 2014b. Discrete stochastic microfacet models. *ACM Transactions on Graphics* 33(4), 115:1–115:10.
- Jakob, W., and S. Marschner. 2012. Manifold exploration: a Markov chain Monte Carlo technique for rendering scenes with difficult specular transport. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2012)* 31(4), 58:1–58:13.
- Jakob, W., C. Regg, and W. Jarosz. 2011. Progressive expectation-maximization for hierarchical volumetric photon mapping. *Computer Graphics Forum (Proceedings of the 2011 Eurographics Symposium on Rendering)* 30(4), 1287–97.
- Jansen, F. W. 1986. Data structures for ray tracing. In L. R. A. Kessener, F. J. Peters, and M. L. P. Lierop (Eds.), *Data Structures for Raster Graphics, Workshop Proceedings*, 57–73. New York: Springer-Verlag.
- Jarabo, A., J. Marco, A. Muñoz, R. Buisan, W. Jarosz, and D. Gutierrez. 2014a. A framework for transient rendering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2014)* 33(6), 177:1–177:10.
- Jarabo, A., H. Wu, J. Dorsey, H. Rushmeier, and D. Gutierrez. 2014b. Effects of approximate filtering on the appearance of bidirectional texture functions. *IEEE Transactions on Visualization and Computer Graphics* 20(6), 880–92.
- Jarosz, W. 2008. Efficient Monte Carlo methods for light transport in scattering media. Ph.D. Thesis, UC San Diego.
- Jarosz, W., D. Nowrouzezahrai, I. Sadeghi, and H. W. Jensen. 2011a. A comprehensive theory of volumetric radiance estimation using photon points and beams. *ACM Transactions on Graphics* 30(1), 5:1–5:19.

- Jarosz, W., D. Nowrouzezahrai, R. Thomas, P.-P. Sloan, and M. Zwicker. 2011b. Progressive photon beams. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2011)* 30(6), 181:1–181:12.
- Jarosz, W., M. Zwicker, and H. W. Jensen. 2008. The beam radiance estimate for volumetric photon mapping. *Computer Graphics Forum (Proceedings of Eurographics 2008)* 27(2), 557–66.
- Jensen, H. W. 1995. Importance driven path tracing using the photon map. In *Eurographics Rendering Workshop 1995*, 326–35.
- Jensen, H. W. 1996. Global illumination using photon maps. In X. Pueyo and P. Schröder (Eds.), *Eurographics Rendering Workshop 1996*, 21–30.
- Jensen, H. W. 1997. Rendering caustics on non-Lambertian surfaces. *Computer Graphics Forum* 16(1), 57–64.
- Jensen, H. W. 2001. *Realistic Image Synthesis Using Photon Mapping*. Natick, Massachusetts: A. K. Peters.
- Jensen, H. W., and J. Buhler. 2002. A rapid hierarchical rendering technique for translucent materials. *ACM Transactions on Graphics* 21(3), 576–81.
- Jensen, H. W., and N. Christensen. 1995. Optimizing path tracing using noise reduction filters. In *Proceedings of WSCG*, 134–42.
- Jensen, H. W., and P. H. Christensen. 1998. Efficient simulation of light transport in scenes with participating media using photon maps. In *SIGGRAPH '98 Conference Proceedings*, Annual Conference Series, 311–20.
- Jensen, H. W., J. Arvo, M. Fajardo, P. Hanrahan, D. Mitchell, M. Pharr, and P. Shirley. 2001a. State of the art in Monte Carlo ray tracing for realistic image synthesis. In *SIGGRAPH 2001 Course 29*, Los Angeles.
- Jensen, H. W., S. R. Marschner, M. Levoy, and P. Hanrahan. 2001b. A practical model for subsurface light transport. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, 511–18.
- Jensen, H. W., J. Arvo, P. Dutré, A. Keller, A. Owen, M. Pharr, and P. Shirley. 2003. Monte Carlo ray tracing. In *SIGGRAPH 2003 Course*, San Diego.
- Jevans, D., and B. Wyvill. 1989. Adaptive voxel subdivision for ray tracing. In *Proceedings of Graphics Interface 1989*, 164–72.
- Joe, S., and F.-Y. Kuo. 2008. Constructing Sobol' sequences with better two-dimensional projections. *SIAM J. Sci. Comput.* 30, 2635–54.
- Johnson, G. M., and M. D. Fairchild. 1999. Full spectral color calculations in realistic image synthesis. *IEEE Computer Graphics and Applications* 19(4), 47–53.
- Johnson, M. K., F. Cole, A. Raj, and E. H. Adelson. 2011. Microgeometry capture using an elastomeric sensor. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2011)* 30(4), 46:1–46:8.
- Johnstone, M. S., and P. R. Wilson. 1999. The memory fragmentation problem: solved? *ACM SIGPLAN Notices* 34(3), 26–36.
- Jones, T. 2005. Efficient generation of Poisson-disk sampling patterns. *Journal of Graphics Tools* 11(2), 27–36.
- Judd, D. B., D. L. MacAdam, and G. Wyszecki. 1964. Spectral distribution of typical daylight as a function of correlated color temperature. *Journal of the Optical Society of America* 54(8), 1031–40.

- Kainz, F., R. Bogart, and D. Hess. 2004. In R. Fernando (Ed.), *GPU Gems*. Reading, Massachusetts: Addison-Wesley, 425–44.
- Kajiya, J. T. 1982. Ray tracing parametric patches. In *Computer Graphics (SIGGRAPH 1982 Conference Proceedings)*, 245–54.
- Kajiya, J. T. 1983. New techniques for ray tracing procedurally defined objects. In *Computer Graphics (Proceedings of SIGGRAPH '83)*, 17, 91–102.
- Kajiya, J. T. 1985. Anisotropic reflection models. *Computer Graphics (Proceedings of SIGGRAPH '85)*, 19, 15–21.
- Kajiya, J. T. 1986. The rendering equation. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, 20, 143–50.
- Kajiya, J. T., and T. L. Kay. 1989. Rendering fur with three dimensional textures. *Computer Graphics (Proceedings of SIGGRAPH '89)*, 23, 271–80.
- Kajiya, J., and M. Ullner. 1981. Filtering high quality text for display on raster scan devices. In *Computer Graphics (Proceedings of SIGGRAPH '81)*, 7–15.
- Kajiya, J. T., and B. P. Von Herzen. 1984. Ray tracing volume densities. In *Computer Graphics (Proceedings of SIGGRAPH '84)*, Volume 18, 165–74.
- Kalantari, N. K., S. Bako, and P. Sen. 2015. A machine learning approach for filtering Monte Carlo noise. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2015)* 34(4), 122:1–122:12.
- Kalantari, N. K., and P. Sen. 2013. Removing the noise in Monte Carlo rendering with general image denoising algorithms. *Computer Graphics Forum (Proceedings of Eurographics 2013)* 32(2), 93–102.
- Kalos, M. H., and P. A. Whitlock. 1986. *Monte Carlo Methods: Volume I: Basics*. New York: Wiley.
- Kalra, D., and A. H. Barr. 1989. Guaranteed ray intersections with implicit surfaces. In *Computer Graphics (Proceedings of SIGGRAPH '89)*, Volume 23, 297–306.
- Kammaje, R., and B. Mora. 2007. A study of restricted BSP trees for ray tracing. In *IEEE Symposium on Interactive Ray Tracing*, 55–62.
- Kaplanyan, A. S., and C. Dachsbacher. 2013a. Adaptive progressive photon mapping. *ACM Transactions on Graphics* 32(2), 16:1–16:13.
- Kaplanyan, A. S., and C. Dachsbaucher. 2013b. Path space regularization for holistic and robust light transport. *Computer Graphics Forum (Proceedings of Eurographics 2013)* 32(2), 63–72.
- Kaplanyan, A. S., J. Hanika, and C. Dachsbaucher. 2014. The natural-constraint representation of the path space for efficient light transport simulation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014)* 33(4), 102:1–102:13.
- Kaplan, M. R. 1985. The uses of spatial coherence in ray tracing. In *ACM SIGGRAPH Course Notes 11*.
- Karras, T., and T. Aila. 2013. Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of High Performance Graphics 2013*, 89–99.
- Karrenberg, R., D. Rubinstein, P. Slusallek, and S. Hack. 2010. AnySL: efficient and portable shading for ray tracing. In *Proceedings of High Performance Graphics 2010*, 97–105.
- Kay, D. S., and D. P. Greenberg. 1979. Transparency for computer synthesized images. In *Computer Graphics (SIGGRAPH '79 Proceedings)*, Volume 13, 158–64.

- Kay, T., and J. Kajiya. 1986. Ray tracing complex scenes. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, Volume 20, 269–78.
- Kelemen, C., L. Szirmay-Kalos, G. Antal, and F. Csonka. 2002. A simple and robust mutation strategy for the Metropolis light transport algorithm. *Computer Graphics Forum* 21(3), 531–40.
- Keller, A. 1996. Quasi-Monte Carlo radiosity. In X. Pueyo and P. Schröder (Eds.), *Eurographics Rendering Workshop 1996*, 101–10.
- Keller, A. 1997. Instant radiosity. In *Proceedings of SIGGRAPH '97*, Computer Graphics Proceedings, Annual Conference Series, Los Angeles, 49–56.
- Keller, A. 1998. Quasi-Monte Carlo methods for photorealistic image synthesis. Ph.D. thesis, Shaker Verlag Aachen.
- Keller, A. 2001. Strictly deterministic sampling methods in computer graphics. *mental images Technical Report*. Also in *SIGGRAPH 2003 Monte Carlo Course Notes*.
- Keller, A. 2004. Stratification by rank-1 lattices. *Monte Carlo and Quasi-Monte Carlo Methods 2002*. Berlin: Springer-Verlag.
- Keller, A. 2006. Myths of computer graphics. In *Monte Carlo and Quasi-Monte Carlo Methods 2004*, Berlin: Springer-Verlag, 217–43.
- Keller, A. 2012. Quasi-Monte Carlo image synthesis in a nutshell. In *Monte Carlo and Quasi-Monte Carlo Methods 2012*, Berlin: Springer-Verlag.
- Keller, A., and C. Wächter. 2011. Efficient ray tracing without auxiliary acceleration data structure. *High Performance Graphics 2011 Poster*.
- Kensler, A., and P. Shirley. 2006. Optimizing ray-triangle intersection via automated search. In *IEEE Symposium on Interactive Ray Tracing*, 33–38.
- Kensler, A. 2008. Tree rotations for improving bounding volume hierarchies. In *IEEE Symposium on Interactive Ray Tracing*, 73–76.
- Kensler, A., A. Knoll, and P. Shirley. 2008. Better gradient noise. *Technical Report UUSCI-2008-001, SCI Institute, University of Utah*.
- Kensler, A. 2013. Correlated multi-jittered sampling. *Pixar Technical Memo 13-01*.
- Kettunen, M., M. Manzi, M. Aittala, J. Lehtinen, F. Durand, and M. Zwicker. 2015. Gradient-domain path tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2015)* 34(4), 123:1–123:13.
- Kider Jr., J. T., D. Knowlton, J. Newlin, Y. K. Li, and D. P. Greenberg. 2014. A framework for the experimental comparison of solar and skydome illumination. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2014)* 33(6), 180:1–180:12.
- Kienle, A., and M. Patterson. 1997. Improved solutions of the steady-state and the time-resolved diffusion equations for reflectance from a semi-infinite turbid medium. *Journal of the Optical Society of America A* 14(1), 246–54.
- Kim, V. G., Y. Lipman, and T. Funkhouser. 2012. Symmetry-guided texture synthesis and manipulation. *ACM Transactions on Graphics* 31(3), 22:1–22:14.
- King, L. V. 1913. On the scattering and absorption of light in gaseous media, with applications to the intensity of sky radiation. *Philosophical Transactions of the Royal Society of London. Series A. Mathematical and Physical Sciences* 212, 375–433.
- King, A., K. Kulla, A. Conty, and M. Fajardo. 2013. BSSRDF importance sampling. *SIGGRAPH 2013 Talks*.

- Kirk, D., and J. Arvo. 1988. The ray tracing kernel. In *Proceedings of Ausgraph '88*, 75–82.
- Kirk, D. B., and J. Arvo. 1991. Unbiased sampling techniques for image synthesis. *Computer Graphics (SIGGRAPH '91 Proceedings)*, Volume 25, 153–56.
- Klassen, R. V. 1987. Modeling the effect of the atmosphere on light. *ACM Transactions on Graphics* 6(3), 215–37.
- Klimaszewski, K. S., and T. W. Sederberg. 1997. Faster ray tracing using adaptive grids. *IEEE Computer Graphics and Applications* 17(1), 42–51.
- Knaus, C., and M. Zwicker. 2011. Progressive photon mapping: a probabilistic approach. *ACM Transactions on Graphics* 30(3), 25:1–25:13.
- Kniep, S., S. Häring, and M. Magnor. 2009. Efficient and accurate rendering of complex light sources. *Computer Graphics Forum (Proceedings of the 2009 Eurographics Symposium on Rendering)* 28(4), 1073–81.
- Knoll, A., Y. Hijazi, C. D. Hansen, I. Wald, and H. Hagen. 2009. Fast ray tracing of arbitrary implicit surfaces with interval and affine arithmetic. *Computer Graphics Forum* 28(1), 26–40.
- Knuth, D. E. 1984. Literate programming. *The Computer Journal* 27, 97–111. Reprinted in D. E. Knuth, *Literate Programming*, Stanford Center for the Study of Language and Information, 1992.
- Knuth, D. E. 1986. *MetaFont: The Program*. Reading, Massachusetts: Addison-Wesley.
- Knuth, D. E. 1993a. *T_EX: The Program*. Reading, Massachusetts: Addison-Wesley.
- Knuth, D. E. 1993b. *The Stanford GraphBase*. New York: ACM Press and Addison-Wesley.
- Kolb, C., D. Mitchell, and P. Hanrahan. 1995. A realistic camera model for computer graphics. *SIGGRAPH '95 Conference Proceedings*, Annual Conference Series, 317–24.
- Kollig, T., and A. Keller. 2000. Efficient bidirectional path tracing by randomized quasi-Monte Carlo integration. In *Monte Carlo and Quasi-Monte Carlo Methods 2000*, pp. 290–305. Berlin: Springer-Verlag.
- Kollig, T., and A. Keller. 2002. Efficient multidimensional sampling. *Computer Graphics Forum (Proceedings of Eurographics 2002)*, Volume 21, 557–63.
- Kontkanen, J., J. Räsänen, and A. Keller. 2004. Irradiance filtering for Monte Carlo ray tracing. *Monte Carlo and Quasi-Monte Carlo Methods*, 259–72.
- Kopta, D., T. Ize, J. Spjut, E. Brunvand, A. Davis, and A. Kensler. 2012. Fast, effective BVH updates for animated scenes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 197–204.
- Křivánek, J., P. Gautron, S. Pattanaik, and K. Bouatouch. 2005. Radiance caching for efficient global illumination computation. *IEEE Transactions on Visualization and Computer Graphics* 11(5), 550–61.
- Kulla, C., and M. Fajardo. 2012. Importance sampling techniques for path tracing in participating media. *Computer Graphics Forum (Proceedings of the 2012 Eurographics Symposium on Rendering)* 31(4), 1519–28.
- Kurt, M., L. Szirmay-Kalos, and J. Křivánek. 2010. An anisotropic BRDF model for fitting and Monte Carlo rendering. *SIGGRAPH Computer Graphics* 44(1), 3:1–3:15.
- Lacewell, D., B. Burley, S. Boulos, and P. Shirley. 2008. Raytracing prefiltered occlusion for aggregate geometry. In *IEEE Symposium on Interactive Ray Tracing*, 19–26.
- Lafortune, E. 1996. Mathematical models and Monte Carlo algorithms for physically based rendering. Ph.D. thesis, Katholieke Universiteit Leuven.

- Lafortune, E., and Y. Willem. 1994. A theoretical framework for physically based rendering. *Computer Graphics Forum* 13(2), 97–107.
- Lafortune, E. P., and Y. D. Willem. 1996. Rendering participating media with bidirectional path tracing. In *Eurographics Rendering Workshop 1996*, 91–100.
- Lagae, A., and G. Drettakis. 2011. Filtering solid Gabor noise. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2011)* 30(4), 51:1–51:6.
- Lagae, A., and P. Dutré. 2005. An efficient ray-quadrilateral intersection test. *Journal of Graphics Tools* 10(4), 23–32.
- Lagae, A., and P. Dutré. 2008a. Compact, fast, and robust grids for ray tracing. In *Computer Graphics Forum (Proceedings of the 2008 Eurographics Symposium on Rendering)* 27(4), 1235–1244.
- Lagae, A., and P. Dutré. 2008b. Accelerating ray tracing using constrained tetrahedralizations. In *Computer Graphics Forum (Proceedings of the 2008 Eurographics Symposium on Rendering)* 27(4), 1303–12.
- Lagae, E., and P. Dutré. 2008c. A comparison of methods for generating Poisson disk distributions. *Computer Graphics Forum* 27(1), 114–29.
- Lagae, A., S. Lefebvre, G. Drettakis, and P. Dutré. 2009. Procedural noise using sparse Gabor convolution. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2009)* 28(3), 54:1–54:10.
- Lagae, A., S. Lefebvre, R. Cook, T. DeRose, G. Drettakis, D. S. Ebert, J. P. Lewis, K. Perlin, and M. Zwicker. 2010. A survey of procedural noise functions. *Computer Graphics Forum* 29(8), 2579–2600.
- Laine, S. 2010. Restart trail for stackless BVH traversal. In *Proceedings of High Performance Graphics 2010*, 107–11.
- Lam, M. S., E. E. Rothberg, and M. E. Wolf. 1991. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Palo Alto, California.
- Lambert, J. H. 1760. *Photometry, or, On the Measure and Gradations of Light, Colors, and Shade*. The Illuminating Engineering Society of North America. Translated by David L. DiLaura in 2001.
- Lang, S. 1986. *An Introduction to Linear Algebra*. New York: Springer-Verlag.
- Lansdale, R. C. 1991. Texture mapping and resampling for computer graphics. M.S. thesis, Department of Electrical Engineering, University of Toronto.
- Larson, G. W., and R. A. Shakespeare. 1998. *Rendering with Radiance: The Art and Science of Lighting Visualization*. San Francisco: Morgan Kaufmann.
- Lauterbach, C., M. Garland, S. Sengupta, D. Luebke, and D. Manocha. 2009. Fast BVH construction on GPUs. *Computer Graphics Forum (Eurographics 2009 Conference Proceedings)* 28(2), 422–30.
- Lawrence, J., S. Rusinkiewicz, and R. Ramamoorthi. 2004. Efficient BRDF importance sampling using a factored representation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004)* 23(3), 496–505.
- Lawrence, J., S. Rusinkiewicz, and R. Ramamoorthi. 2005. Adaptive numerical cumulative distribution functions for efficient importance sampling. *Rendering Techniques 2005: 16th Eurographics Workshop on Rendering*, 11–20.

- L'Ecuyer, P., and R Simard. 2007. TestU01: a C library for empirical testing of random number generators. In *ACM Transactions on Mathematical Software* 33(4).
- Lee, M. E., R. A. Redner, and S. P. Uselton. 1985. Statistically optimized sampling for distributed ray tracing. In *Computer Graphics (Proceedings of SIGGRAPH '85)*, Volume 19, 61–67.
- Lee, M., and R. Redner. 1990. A note on the use of nonlinear filtering in computer graphics. *IEEE Computer Graphics and Applications* 10(3), 23–29.
- Lee, W.-J., Y. Shin, J. Lee, J.-W. Kim, J.-H. Nah, S. Jung, S. Lee, H.-S. Park, and T.-D. Han. 2013. SGRT: a mobile GPU architecture for real-time ray tracing. In *Proceedings of High Performance Graphics 2013*, 109–19.
- Lee, W.-J., Y. Shin, S. J. Hwang, S. Kang, J.-J. Yoo, and S. Ryu. 2015. Reorder buffer: an energy-efficient multithreading architecture for hardware SIMD ray traversal. In *Proceedings of High Performance Graphics 2015*, 21–32.
- Lefebvre, S., S. Hornus, and A. Lasram. 2010. By-example synthesis of architectural textures. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010)* 29(4), 84:1–84:8.
- Lehtinen, J., T. Aila, J. Chen, S. Laine, and F. Durand. 2011. Temporal light field reconstruction for rendering distribution effects. *ACM SIGGRAPH 2011 Papers*. 55:1–55:12.
- Lehtinen, J., T. Aila, S. Laine, and F. Durand. 2012. Reconstructing the indirect light field for global illumination. *ACM Transactions on Graphics* 31(4). 51:1–51:10.
- Lehtinen, J., T. Karras, S. Laine, M. Aittala, F. Durand, and T. Aila. 2013. Gradient-domain Metropolis light transport. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2013)* 32(4), 95:1–95:12.
- Lehtonen, J., J. Parkkinen, and T. Jaaskelainen. 2006. Optimal sampling of color spectra. *Journal of the Optical Society of America A* 23(13), 2983–88.
- Lessig, C., M. Desbrun, and E. Fiume. 2014. A constructive theory of sampling for image synthesis using reproducing kernel bases. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014)* 33(4), 55:1–55:14.
- Levine, J. R., T. Mason, and D. Brown. 1992. *lex & yacc*. Sebastopol, California: O'Reilly & Associates.
- Levoy, M. 1988. Display of surfaces from volume data. *IEEE Computer Graphics and Applications* 8(3), 29–37.
- Levoy, M. 1990a. Efficient ray tracing of volume data. *ACM Transactions on Graphics* 9(3), 245–61.
- Levoy, M. 1990b. A hybrid ray tracer for rendering polygon and volume data. *IEEE Computer Graphics and Applications* 10(2), 33–40.
- Levoy, M., and P. M. Hanrahan. 1996. Light field rendering. In *Proceedings of SIGGRAPH '96*, Computer Graphics Proceedings, Annual Conference Series, 31–42.
- Levoy, M., and T. Whitted. 1985. The use of points as a display primitive. *Technical Report 85-022*. Computer Science Department, University of North Carolina at Chapel Hill.
- Lewis, J.-P. 1989. Algorithms for solid noise synthesis. In *Computer Graphics (Proceedings of SIGGRAPH '89)*, Volume 23, 263–70.
- Li, H., L.-Y. Wei, P. Sander, and C.-W. Fu. 2010. Anisotropic blue noise sampling. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2010)* 29(6), 167:1–167:12.

- Li, K., F. Pellacini, and K. Torrance. 2005. A hybrid Monte Carlo method for accurate and efficient subsurface scattering. In *Rendering Techniques (Proceedings of the 2005 Eurographics Symposium on Rendering)*, 283–90.
- Liu, J. S. 2001. *Monte Carlo Strategies in Scientific Computing*. New York: Springer-Verlag.
- Logie, J. R., and J. W. Patterson. 1994. Inverse displacement mapping in the general case. *Computer Graphics Forum* 14(5), 261–73.
- Lokovic, T., and E. Veach. 2000. Deep shadow maps. In *Proceedings of ACM SIGGRAPH 2000, Computer Graphics Proceedings, Annual Conference Series*, 385–92.
- Lommel, E. 1889. Die Photometrie der diffusen Zurückwerfung. *Annalen der Physik* 36, 473–502.
- Loop, C. 1987. Smooth subdivision surfaces based on triangles. M.S. thesis, University of Utah.
- Lu, H., R. Paganowski, and X. Granier. 2013. Second-order approximation for variance reduction in multiple importance sampling. *Computer Graphics Forum* 32(7), 131–36.
- Lu, H., R. Paganowski, and X. Granier. 2015. Position-dependent importance sampling of light field luminaires. *IEEE Transactions on Visualization and Computer Graphics* 21(2), 241–51.
- Lukaszewski, A. 2001. Exploiting coherence of shadow rays. In *AFRIGRAPH 2001*, 147–50. ACM SIGGRAPH.
- MacDonald, J. D., and K. S. Booth. 1990. Heuristics for ray tracing using space subdivision. *The Visual Computer* 6(3), 153–66.
- Machiraju, R., and R. Yagel. 1996. Reconstruction error characterization and control: a sampling theory approach. *IEEE Transactions on Visualization and Computer Graphics* 2(4).
- MacKay, D. 2003. *Information Theory, Inference, and Learning Algorithms*. Cambridge: Cambridge University Press.
- Malacara, D. 2002. *Color Vision and Colorimetry: Theory and Applications*. SPIE—The International Society for Optical Engineering.
- Mann, S., N. Litke, and T. DeRose. 1997. A coordinate free geometry ADT. Research Report CS-97-15, Computer Science Department, University of Waterloo.
- Manson, J., and S. Schaefer. 2013. Cardinality-constrained texture filtering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2013)* 32(4), 140:1–140:8.
- Manson, J., and S. Schaefer. 2014. Bilinear accelerated filter approximation. *Computer Graphics Forum (Proceedings of the 2014 Eurographics Symposium on Rendering)* 33(4), 33–40.
- Mansson, E., J. Munkberg, and T. Akenine-Möller. 2007. Deep coherent ray tracing. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, 79–85.
- Manzi, M., M. Kettunen, M. Aittala, J. Lehtinen, F. Durand, and M. Zwicker. 2015. Gradient-domain bidirectional path tracing. *Eurographics Symposium on Rendering—Experimental Ideas & Implementations*.
- Manzi, M., F. Rousselle, M. Kettunen, J. Lehtinen, and M. Zwicker. 2014. Improved sampling for gradient-domain Metropolis light transport. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2014)* 33(6), 178:1–178:12.
- Marques, R., C. Bouville, M. Ribardière, L. P. Santos, and K. Bouatouch. 2013. Spherical Fibonacci point sets for illumination integrals. *Computer Graphics Forum (Proceedings of the 2013 Eurographics Symposium on Rendering)* 32(4), 134–43.

- Marschner, S. 1998. Inverse rendering for computer graphics. Ph.D. thesis, Cornell University.
- Marschner, S. R., H. W. Jensen, M. Cammarano, S. Worley, and P. Hanrahan. 2003. Light scattering from human hair fibers. *ACM Transactions on Graphics* 22(3), 780–91.
- Marschner, S. R., and R. J. Lobb. 1994. An evaluation of reconstruction filters for volume rendering. In *Proceedings of Visualization '94*, Washington, D.C., 100–07.
- Marschner, S., S. Westin, A. Arbree, and J. Moon. 2005. Measuring and modeling the appearance of finished wood. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)* 24(3), 727–34.
- Martin, W., E. Cohen, R. Fish, and P. S. Shirley. 2000. Practical ray tracing of trimmed NURBS surfaces. *Journal of Graphics Tools* 5(1), 27–52.
- Mas, A., I. Martín, and G. Patow. 2008. Compression and importance sampling of near-field light sources. *Computer Graphics Forum* 27(8), 2013–27.
- Matusik, W., H. Pfister, M. Brand, and L. McMillan. 2003a. Efficient isotropic BRDF measurement. In *Proceedings of the 14th Eurographics Workshop on Rendering*, 241–47.
- Matusik, W., H. Pfister, M. Brand, and L. McMillan. 2003b. A data-driven reflectance model. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003)* 22(3), 759–69.
- Max, N. L. 1986. Atmospheric illumination and shadows. In *Computer Graphics (Proceedings of SIGGRAPH '86)*, Volume 20, 117–24.
- Max, N. L. 1988. Horizon mapping: shadows for bump-mapped surfaces. *The Visual Computer* 4(2), 109–17.
- Max, N. L. 1995. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics* 1(2), 99–108.
- McCluney, W. R. 1994. *Introduction to Radiometry and Photometry*. Boston: Artech House.
- McCombe, J. 2013. Low power consumption ray tracing. *SIGGRAPH 2013 Course: Ray Tracing Is the Future and Ever Will Be*.
- McCormack, J., R. Perry, K. I. Farkas, and N. P. Jouppi. 1999. Feline: fast elliptical lines for anisotropic texture mapping. In *Proceedings of SIGGRAPH '99*, Computer Graphics Proceedings, Annual Conference Series, Los Angeles, 243–250.
- Meijering, E. 2002. A chronology of interpolation: from ancient astronomy to modern signal and image processing. In *Proceedings of the IEEE* 90(3), 319–42.
- Meijering, E. H. W., W. J. Niessen, J. P. W. Pluim, and M. A. Viergever. 1999. Quantitative comparison of sinc-approximating kernels for medical image interpolation. In C. Taylor and A. Colchester (Eds.), *Medical Image Computing and Computer-Assisted Intervention—MICCAI 1999*, 210–17. Berlin: Springer-Verlag.
- Meng, J., F. Simon, J. Hanika, and C. Dachsbacher. 2015. Physically meaningful rendering using tristimulus colours. *Computer Graphics Forum (Proceedings of the 2015 Eurographics Symposium on Rendering)* 34(4), 31–40.
- Metropolis, N., A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. 1953. Equation of state calculations by fast computing machines. *Journal of Chemical Physics* 21(6), 1087–92.
- Meyer, G. W., and D. P. Greenberg. 1980. Perceptual color spaces for computer graphics. In *Computer Graphics (Proceedings of SIGGRAPH '80)*, Volume 14, Seattle, Washington, 254–261.
- Meyer, G. W., H. E. Rushmeier, M. F. Cohen, D. P. Greenberg, and K. E. Torrance. 1986. An experimental evaluation of computer graphics imagery. *ACM Transactions on Graphics* 5(1), 30–50.

- Mikkelsen, M. 2008. Simulation of wrinkled surfaces revisited. M.S. thesis, University of Copenhagen.
- Miller, G. S., and C. R. Hoffmann. 1984. Illumination and reflection maps: simulated objects in simulated and real environments. *Course Notes for Advanced Computer Graphics Animation, SIGGRAPH '84*.
- Mitchell, D. P. 1987. Generating antialiased images at low sampling densities. *Computer Graphics (SIGGRAPH '87 Proceedings)*, Volume 21, 65–72.
- Mitchell, D. P. 1990. Robust ray intersection with interval arithmetic. In *Proceedings of Graphics Interface 1990*, 68–74.
- Mitchell, D. P. 1991. Spectrally optimal sampling for distributed ray tracing. *Computer Graphics (SIGGRAPH '91 Proceedings)*, Volume 25, 157–64.
- Mitchell, D. P. 1992. Ray tracing and irregularities of distribution. In *Third Eurographics Workshop on Rendering*, Bristol, United Kingdom, 61–69.
- Mitchell, D. P. 1996b. Consequences of stratified sampling in graphics. In *Proceedings of SIGGRAPH '96*, Computer Graphics Proceedings, Annual Conference Series, New Orleans, Louisiana, 277–80.
- Mitchell, D. P., and P. Hanrahan. 1992. Illumination from curved reflectors. In *Computer Graphics (Proceedings of SIGGRAPH '92)*, Volume 26, 283–91.
- Mitchell, D. P., and A. N. Netravali. 1988. Reconstruction filters in computer graphics. *Computer Graphics (SIGGRAPH '88 Proceedings)*, Volume 22, 221–28.
- Möller, T., R. Machiraju, K. Mueller, and R. Yagel. 1997. Evaluation and design of filters using a Taylor series expansion. *IEEE Transactions on Visualization and Computer Graphics* 3(2), 184–99.
- Möller, T., and B. Trumbore. 1997. Fast, minimum storage ray–triangle intersection. *Journal of Graphics Tools* 2(1), 21–28.
- Moon, B., Y. Byun, T.-J. Kim, P. Claudio, H.-S. Kim, Y.-J. Ban, S. W. Nam, and S.-E. Yoon. 2010. Cache-oblivious ray reordering. *ACM Transactions on Graphics* 29(3), 28:1–28:10.
- Moon, B., N. Carr, and S.-E. Yoon. 2014. Adaptive rendering based on weighted local regression. *ACM Transactions on Graphics* 33(5), 170:1–170:14.
- Moon, J., and S. Marschner. 2006. Simulating multiple scattering in hair using a photon mapping approach. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006)* 25(3), 1067–74.
- Moon, J., B. Walter, and S. Marschner. 2007. Rendering discrete random media using precomputed scattering solutions. *Rendering Techniques 2007: 18th Eurographics Workshop on Rendering*, 231–42.
- Moon, J., B. Walter, and S. Marschner. 2008. Efficient multiple scattering in hair using spherical harmonics. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008)* 27(3), 31:1–31:7.
- Moon, P., and D. E. Spencer. 1936. *The Scientific Basis of Illuminating Engineering*. New York: McGraw-Hill.
- Moon, P., and D. E. Spencer. 1948. *Lighting Design*. Reading, Massachusetts: Addison-Wesley.
- Moore, R. E. 1966. *Interval Analysis*. Englewood Cliffs, New Jersey: Prentice Hall.
- Mora, B. 2011. Naive ray-tracing: A divide-and-conquer approach. *ACM Transactions on Graphics* 30(5), 117:1–117:12.

- Moravec, H. 1981. 3D graphics and the wave theory. In *Computer Graphics*, Volume 15, 289–96.
- Morley, R. K., S. Boulos, J. Johnson, D. Edwards, P. Shirley, M. Ashikhmin, and S. Premonoze. 2006. Image synthesis using adjoint photons. In *Proceedings of Graphics Interface 2006*, 179–86.
- Motwani, R., and P. Raghavan. 1995. *Randomized Algorithms*. Cambridge, U.K.: Cambridge University Press.
- Moulin, M., N. Billen, P. Dutré. 2015. Efficient visibility heuristics for kd-trees using the RTSAH. *Eurographics Symposium on Rendering—Experimental Ideas & Implementations*.
- Moulton, J. 1990. Diffusion modeling of picosecond laser pulse propagation in turbid media. Master’s thesis, McMaster University.
- Müller, K., T. Techmann, and D. Fellner. 2003. Adaptive ray tracing of subdivision surfaces. *Computer Graphics Forum* 22(3), 553–62.
- Müller, G., J. Meseth, M. Sattler, R. Sarlette, and R. Klein. 2005. Acquisition, synthesis and rendering of bidirectional texture functions. *Computer Graphics Forum (Eurographics State of the Art Report)* 24(1), 83–109.
- Munkberg, J., K. Vaidyanathan, J. Hasselgren, P. Clarberg, and T. Akenine-Möller. 2014. Layered reconstruction for defocus and motion blur. *Computer Graphics Forum* 33, 81–92.
- Musbach, A., G. W. Meyer, F. Reitich, and S. H. Oh. 2013. Full wave modelling of light propagation and reflection. *Computer Graphics Forum* 32(6), 24–37.
- Museth, K. 2013. VDB: high-resolution sparse volumes with dynamic topology. *ACM Transactions on Graphics* 32(3), 27:1–27:22.
- Musgrave, K. 1992. A panoramic virtual screen for ray tracing. In D. Kirk (Ed.), *Graphics Gems III*, 288–94. San Diego: Academic Press.
- Nabata, K., K. Iwasaki, Y. Dobashi, and T. Nishita. 2013. Efficient divide-and-conquer ray tracing using ray sampling. In *Proceedings of High Performance Graphics 2013*, 129–35.
- Nah, J.-H., J.-S. Park, C. Park, J.-W. Kim, Y.-H. Jung, W.-C. Park, and T.-D. Han. 2011. T&I engine: traversal and intersection engine for hardware accelerated ray tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2011)* 30(6), 160:1–160:10.
- Nakamaru, K., and Y. Ohno. 2002. Ray tracing for curves primitive. In *Journal of WSCG (WSCG 2002 Proceedings)* 10, 311–16.
- Narasimhan, S., M. Gupta, C. Donner, R. Ramamoorthi, S. Nayar, and H. W. Jensen. 2006. Acquiring scattering properties of participating media by dilution. *ACM Transactions on Graphics* 25(3), 1003–12.
- Nayar, S. K., K. Ikeuchi, and T. Kanade. 1991. Surface reflection: physical and geometrical perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 17(7), 611–34.
- Naylor, B. 1993. Constructing good partition trees. In *Proceedings of Graphics Interface 1993*, 181–91.
- Neyret, F. 1996. Synthesizing verdant landscapes using volumetric textures. In *Eurographics Rendering Workshop 1996*, 215–24.
- Neyret, F. 1998. Modeling, animating, and rendering complex scenes using volumetric textures. *IEEE Transactions on Visualization and Computer Graphics* 4(1), 55–70.

- Nicodemus, F., J. Richmond, J. Hsia, I. Ginsburg, and T. Limperis. 1977. *Geometrical Considerations and Nomenclature for Reflectance*. NBS Monograph 160, Washington, D.C.: National Bureau of Standards, U.S. Department of Commerce.
- Niederreiter, H. 1992. *Random Number Generation and Quasi-Monte Carlo Methods*. Philadelphia: Society for Industrial and Applied Mathematics.
- Nishita, T., and E. Nakamae. 1985. Continuous tone representation of three-dimensional objects taking account of shadows and interreflection. *SIGGRAPH Computer Graphics* 19(3), 23–30.
- Nishita, T., and E. Nakamae. 1986. Continuous tone representation of three-dimensional objects illuminated by sky light. In *Computer Graphics (Proceedings of SIGGRAPH '86)*, Volume 20, 125–32.
- Nishita, T., Y. Miyawaki, and E. Nakamae. 1987. A shading model for atmospheric scattering considering luminous intensity distribution of light sources. In *Computer Graphics (Proceedings of SIGGRAPH '87)*, Volume 21, 303–10.
- Ngan, A., F. Durand, and W. Matusik. 2005. Experimental analysis of BRDF models. *Rendering Techniques 2005: 16th Eurographics Workshop on Rendering*, 117–26.
- Ng, R., M. Levoy, M. Brédif, G. Duval, M. Horowitz, and P. Hanrahan. 2005. Light field photography with a hand-held plenoptic camera. *Stanford University Computer Science Technical Report*, CSTR 2005-02.
- Norton, A., A. P. Rockwood, and P. T. Skolmoski. 1982. Clamping: a method of antialiasing textured surfaces by bandwidth limiting in object space. In *Computer Graphics (Proceedings of SIGGRAPH '82)*, Volume 16, 1–8.
- Novák, J., A. Selle, and W. Jarosz. 2014. Residual ratio tracking for estimating attenuation in participating media. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2014)* 33(6), 179:1–179:11.
- O'Neill, M. 2014. PCG: A family of simple fast space-efficient statistically good algorithms for random number generation. Unpublished manuscript. <http://www.pcg-random.org/paper.html>.
- Ogaki, S., and Y. Tokuyoshi. 2011. Direct ray tracing of Phong tessellation. *Computer Graphics Forum (Proceedings of the 2011 Eurographics Symposium on Rendering)* 30(4), 1337–44.
- Ohmer, S. 1997. Ray Tracers: Blue Sky Studios. *Animation World Network*, <http://www.awn.com/animationworld/ray-tracers-blue-sky-studios>.
- Olano, M., and D. Baker. 2010. LEAN mapping. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, 181–88.
- OpenMP Architecture Review Board. 2013. OpenMP Application Program Interface. <http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>.
- Ooi, B. C., K. McDonell, and R. Sacks-Davis. 1987. Spatial kd-tree: a data structure for geographic databases. In *Proceedings of the IEEE COMPSAC Conference*.
- Oren, M., and S. K. Nayar. 1994. Generalization of Lambert's reflectance model. In *Proceedings of SIGGRAPH '94, Computer Graphics Proceedings, Annual Conference Series*, 239–46. New York: ACM Press.
- Ou, J., and F Pellacini. 2010. SafeGI: type checking to improve correctness in rendering system implementation. *Computer Graphics Forum (Proceedings of the 2010 Eurographics Symposium on Rendering)* 29(4), 1267–77.

- Ou, J., F. Xie, P. Krishnamachari, and F. Pellacini. 2012. ISHair: importance sampling for hair scattering. *Computer Graphics Forum (Proceedings of the 2012 Eurographics Symposium on Rendering)* 31(4), 1537–45.
- Overbeck, R., C. Donner, and R. Ramamoorthi. 2009. Adaptive wavelet rendering. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2009)* 28(5), 140:1–140:12.
- Owen, A. B. 1998. Latin supercube sampling for very high-dimensional simulations. *Modeling and Computer Simulation* 8(1), 71–102.
- Pacanowski, R., O. Salazar-Celis, C. Schlick, X. Granier, P. Poulin, and A. Cuyt. 2012. Rational BRDF. *IEEE Transactions on Visualization and Computer Graphics* 18(11), 1824–35.
- Pajot, A., L. Barthe, M. Paulin, and P. Poulin. 2011. Representativity for robust and adaptive multiple importance sampling. *IEEE Transactions on Visualization and Computer Graphics* 17(8), 1108–21.
- Pantaleoni, J., L. Fascione, M. Hill, and T. Aila. 2010. PantaRay: fast ray-traced occlusion caching of massive scenes. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010)* 29(4), 37:1–37:10.
- Pantaleoni, J., and D. Luebke. 2010. HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics 2010*, 87–95.
- Papas, M., K. de Mesa, and H. W. Jensen. 2014. A physically-based BSDF for modeling the appearance of paper. *Computer Graphics Forum (Proceedings of the 2014 Eurographics Symposium on Rendering)* 33(4), 133–42.
- Parker, S., S. Boulos, J. Bigler, and A. Robison. 2007. RTSL: a ray tracing shading language. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*.
- Parker, S. G., J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. 2010. OptiX: a general purpose ray tracing engine. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010)* 29(4), 66:1–66:13.
- Parker, S., W. Martin, P.-P. J. Sloan, P. S. Shirley, B. Smits, and C. Hansen. 1999. Interactive ray tracing. In *1999 ACM Symposium on Interactive 3D Graphics*, 119–26.
- Patney, A., M. S. Ebeida, and J. D. Owens. 2009. Parallel view-dependent tessellation of Catmull–Clark subdivision surfaces. In *Proceedings of High Performance Graphics 2009*, 99–108.
- Pattanaik, S. N., and S. P. Mudur. 1995. Adjoint equations and random walks for illumination computation. *ACM Transactions on Graphics* 14(1), 77–102.
- Patterson, D., and J. Hennessy. 2006. *Computer Architecture: A Quantitative Approach*. San Francisco: Morgan Kaufmann.
- Patterson, J. W., S. G. Hoggard, and J. R. Logie. 1991. Inverse displacement mapping. *Computer Graphics Forum* 10(2), 129–39.
- Pauly, M. 1999. Robust Monte Carlo methods for photorealistic rendering of volumetric effects. Master's thesis, Universität Kaiserslautern.
- Pauly, M., T. Kollig, and A. Keller. 2000. Metropolis light transport for participating media. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, 11–22.
- Peachey, D. R. 1985. Solid texturing of complex surfaces. *Computer Graphics (SIGGRAPH '85 Proceedings)*, Volume 19, 279–86.
- Peachey, D. R. 1990. Texture on demand. Pixar Technical Memo #217.

- Pearce, A. 1991. A recursive shadow voxel cache for ray tracing. In J. Arvo (Ed.), *Graphics Gems II*, 273–74. San Diego: Academic Press.
- Peercy, M. S. 1993. Linear color representations for full spectral rendering. *Computer Graphics (SIGGRAPH '93 Proceedings)*, Volume 27, 191–98.
- Peers, P., K. vom Berge, W. Matusik, R. Ramamoorthi, J. Lawrence, S. Rusinkiewicz, and P. Dutré. 2006. A compact factored representation of heterogeneous subsurface scattering. *ACM Transactions on Graphics* 25(3), 746–53.
- Pegoraro, V., and S. Parker. 2009. An analytical solution to single scattering in homogeneous participating media. *Computer Graphics Forum (Proceedings of Eurographics 2009)* 28(2), 329–35.
- Pegoraro, V., M. Schott, and S. Parker. 2009. An analytical approach to single scattering for anisotropic media and light distributions. In *Proceedings of Graphics Interface 2009*, 71–77.
- Pegoraro, V., C. Brownlee, P. Shirley, and S. Parker. 2008a. Towards interactive global illumination effects via sequential Monte Carlo adaptation. *IEEE Symposium on Interactive Ray Tracing*, 107–14.
- Pegoraro, V., M. Schott, and S. G. Parker. 2010. A closed-form solution to single scattering for general phase functions and light distributions. *Computer Graphics Forum (Proceedings of the 2010 Eurographics Symposium on Rendering)* 29(4), 1365–74.
- Pegoraro, V., M. Schott, and P. Slusallek. 2011. A mathematical framework for efficient closed-form single scattering. In *Proceedings of Graphics Interface 2011*, 151–58.
- Pegoraro, V., I. Wald, and S. Parker. 2008b. Sequential Monte Carlo adaptation in low-anisotropy participating media. *Computer Graphics Forum (Proceedings of the 2008 Eurographics Symposium on Rendering)* 27(4), 1097–1104.
- Pekelis, L., and C. Hery. 2014. A statistical framework for comparing importance sampling methods, and an application to rectangular lights. *Pixar Technical Memo 14-01*.
- Pekelis, L., C. Hery, R. Villemin, and J. Ling. 2015. A data-driven light scattering model for hair. *Pixar Technical Memo 15-02*.
- Perlin, K. 1985a. An image synthesizer. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, Volume 19, 287–96.
- Perlin, K. 1985b. State of the art in image synthesis. *SIGGRAPH Course Notes 11*.
- Perlin, K. 2002. Improving noise. *ACM Transactions on Graphics* 21(3), 681–82.
- Perlin, K., and E. M. Hoffert. 1989. Hypertexture. In *Computer Graphics (Proceedings of SIGGRAPH '89)*, Volume 23, 253–62.
- Pfister, H., M. Zwicker, J. van Baar, and M. Gross. 2000. Surfels: Surface elements as rendering primitives. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, 335–42.
- Pharr, M., and P. Hanrahan. 1996. Geometry caching for ray-tracing displacement maps. In *Eurographics Rendering Workshop 1996*, 31–40.
- Pharr, M., and P. M. Hanrahan. 2000. Monte Carlo evaluation of non-linear scattering equations for subsurface reflection. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, 75–84.
- Pharr, M., and G. Humphreys. 2004. *Physically Based Rendering: From Theory to Implementation*. San Francisco: Morgan Kaufmann.

- Pharr, M., C. Kolb, R. Gershbein, and P. M. Hanrahan. 1997. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of SIGGRAPH '97*, Computer Graphics Proceedings, Annual Conference Series, 101–08.
- Pharr, M., and W. R. Mark. 2012. ispc: a SPMD compiler for high-performance CPU programming. In *Proceedings of Innovative Parallel Computing (InPar)*.
- Phong, B.-T. 1975. Illumination for computer generated pictures. *Communications of the ACM* 18(6), 311–17.
- Phong, B.-T., and F. C. Crow. 1975. Improved rendition of polygonal models of curved surfaces. In *Proceedings of the 2nd USA-Japan Computer Conference*.
- Pilleboue, A., G. Singh, D. Coeurjolly, M. Kazhdan, and V. Ostromoukhov. 2015. Variance analysis for Monte Carlo integration. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2015)* 34(4), 124:1–124:14.
- Piponi, D. 2012. Lossless decompression and the generation of random samples. <http://blog.sigfpe.com/2012/01/lossless-decompression-and-generation.html>.
- Pixar Animation Studios. 2000. The RenderMan Interface. Version 3.2.
- Pomraning, G. C., and B. D. Ganapol. 1995. Asymptotically consistent reflection boundary conditions for diffusion theory. In *Annals of Nuclear Energy* 22(12), 787–817.
- Popov, S., I. Georgiev, P. Slusallek, and C. Dachsbacher. 2013. Adaptive quantization visibility caching. *Computer Graphics Forum (Proceedings of Eurographics 2013)* 32(2), 399–408.
- Popov, S., J. Gunther, H. P. Seidel, and P. Slusallek. 2006. Experiences with streaming construction of SAH kd-trees. In *IEEE Symposium on Interactive Ray Tracing*, 89–94.
- Popov, S., R. Dimov, I. Georgiev, and P. Slusallek. 2009. Object partitioning considered harmful: space subdivision for BVHs. In *Proceedings of High Performance Graphics 2009*, 15–22.
- Porumbescu, S., B. Budge, L. Feng, and K. Joy. 2005. Shell maps. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)* 24(3), 626–33.
- Potmesil, M., and I. Chakravarty. 1981. A lens and aperture camera model for synthetic image generation. In *Computer Graphics (Proceedings of SIGGRAPH '81)*, Volume 15, 297–305.
- Potmesil, M., and I. Chakravarty. 1982. Synthetic image generation with a lens and aperture camera model. *ACM Transactions on Graphics* 1(2), 85–108.
- Potmesil, M., and I. Chakravarty. 1983. Modeling motion blur in computer-generated images. In *Computer Graphics (Proceedings of SIGGRAPH 83)*, Volume 17, Detroit, Michigan, 389–99.
- Poulin, P., and A. Fournier. 1990. A model for anisotropic reflection. In *Computer Graphics (Proceedings of SIGGRAPH '90)*, Volume 24, 273–82.
- Poynton, C. 2002a. Frequently-asked questions about color. www.poynton.com/ColorFAQ.html.
- Poynton, C. 2002b. Frequently-asked questions about gamma. www.poynton.com/GammaFAQ.html.
- Preetham, A. J., P. S. Shirley, and B. E. Smits. 1999. A practical analytic model for daylight. In *Proceedings of SIGGRAPH '99*, Computer Graphics Proceedings, Annual Conference Series, 91–100.
- Preisendorfer, R. W. 1965. *Radiative Transfer on Discrete Spaces*. Oxford: Pergamon Press.
- Preisendorfer, R. W. 1976. *Hydrologic Optics*. Honolulu, Hawaii: U.S. Department of Commerce, National Oceanic and Atmospheric Administration.

- Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. 1992. *Numerical Recipes in C: The Art of Scientific Computing* (2nd ed.). Cambridge: Cambridge University Press.
- Prusinkiewicz, P. 1986. Graphical applications of L-systems. In *Proceedings of Graphics Interface 1986*, 247–53.
- Prusinkiewicz, P., M. James, and R. Mech. 1994. Synthetic topiary. In *Proceedings of SIGGRAPH '94*, Computer Graphics Proceedings, Annual Conference Series, 351–58.
- Prusinkiewicz, P., L. Mündermann, R. Karwowski, and B. Lane. 2001. The use of positional information in the modeling of plants. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, 289–300.
- Purcell, T. J., I. Buck, W. R. Mark, and P. Hanrahan. 2002. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics* 21(3), 703–12.
- Purcell, T. J., C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. 2003. Photon mapping on programmable graphics hardware. In *Graphics Hardware 2003*, 41–50.
- Purgathofer, W. 1987. A statistical method for adaptive stochastic sampling. *Computers & Graphics* 11(2), 157–62.
- Qin, H., M. Chai, Q. Hou, Z. Ren, and K. Zhou. 2014. Cone tracing for furry object rendering. *IEEE Transactions on Visualization and Computer Graphics* 20(8), 1178–88.
- Quilez, I. 2015. Distance estimation. <http://iquilezles.org/www/articles/distance/distance.htm>.
- Raab, M., D. Seibert, and A. Keller. 2006. Unbiased global illumination with participating media. *Proc. Monte Carlo and Quasi-Monte Carlo Methods 2006*, 591–605.
- Radziszewski, M., K. Boryczko, and W. Alda. 2009. An improved technique for full spectral rendering. *Journal of WSCG* 17(1-3), 9–16.
- Ramamoorthi, R., and A. Barr. 1997. Fast construction of accurate quaternion splines. In *Proceedings of SIGGRAPH '97*, Computer Graphics Proceedings, Annual Conference Series, Los Angeles, 287–92.
- Ramsey, S. D., K. Potter, and C. Hansen. 2004. Ray bilinear patch intersections. *Journal of Graphics Tools* 9(3), 41–47.
- Ramshaw, L. 1987. Blossoming: a connect-the-dots approach to splines. *Digital Systems Research Center Technical Report*.
- Reeves, W. T., D. H. Salesin, and R. L. Cook. 1987. Rendering antialiased shadows with depth maps. In *Computer Graphics (Proceedings of SIGGRAPH '87)*, Volume 21, 283–91.
- Reichert, M. C. 1992. A two-pass radiosity method driven by lights and viewer position. Master's thesis, Cornell University.
- Reinert, B., T. Ritschel, H.-P. Seidel, and I. Georgiev. 2015. Projective blue-noise sampling. In *Computer Graphics Forum*.
- Reinhard, E., T. Pouli, T. Kunkel, B. Long, A. Ballestad, and G. Damberg. 2012. Calibrated image appearance reproduction. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2012)* 31(6), 201:1–201:11.
- Reinhard, E., G. Ward, P. Debevec, S. Pattanaik, W. Heidrich, and K. Myszkowski. 2010. *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting*. San Francisco: Morgan Kaufmann.
- Reshetov, A., A. Souppikov, and J. Hurley. 2005. Multi-level ray tracing algorithm. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)* 24(3), 1176–85.

- Reshetov, A. 2007. Faster ray packets–triangle intersection through vertex culling. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, 105–12.
- Reshetov, A. 2009. Morphological antialiasing. In *Proceedings of High Performance Graphics 2009*.
- Rogers, D. F., and J. A. Adams. 1990. *Mathematical Elements for Computer Graphics*. New York: McGraw-Hill.
- Ross, S. M. 2002. *Introduction to Probability Models* (8th ed.). San Diego: Academic Press.
- Roth, S. D. 1982. Ray casting for modeling solids. *Computer Graphics and Image Processing* 18, 109–44.
- Roth, S. H., P. Diezi, and M. Gross. 2001. Ray tracing triangular Bézier patches. In *Computer Graphics Forum (Eurographics 2001 Conference Proceedings)* 20(3), 422–30.
- Rougeron, G., and B. Péroche. 1997. An adaptive representation of spectral data for reflectance computations. In *Eurographics Rendering Workshop 1997*, 126–38.
- Rougeron, G., and B. Péroche. 1998. Color fidelity in computer graphics: a survey. *Computer Graphics Forum* 17(1), 3–16.
- Rousselle, F., P. Clarberg, L. Leblanc, V. Ostromoukhov, and P. Poulin. 2008. Efficient product sampling using hierarchical thresholding. *The Visual Computer (Proceedings of CGI 2008)* 24(7–9), 465–74.
- Rousselle, F., C. Knaus, and M. Zwicker. 2012. Adaptive rendering with non-local means filtering. *ACM Transactions on Graphics* 31(6), 195:1–195:11.
- Rousselle, F., M. Manzi, and M. Zwicker. 2013. Robust denoising using feature and color information. *Computer Graphics Forum (Proceedings of Pacific Graphics)* 32(7), 121–30.
- Rubin, S. M., and T. Whitted. 1980. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics* 14(3), 110–16.
- Ruckert, M. 2005. *Understanding MP3*. Wiesbaden, Germany: GWV-Vieweg.
- Rushmeier, H. E. 1988. Realistic image synthesis for scenes with radiatively participating media. Ph.D. thesis, Cornell University.
- Rushmeier, H. E., and K. E. Torrance. 1987. The zonal method for calculating light intensities in the presence of a participating medium. In *Computer Graphics (Proceedings of SIGGRAPH '87)*, Volume 21, 293–302.
- Rushmeier, H., C. Patterson, and A. Veerasamy. 1993. Geometric simplification for indirect illumination calculations. In *Proceedings of Graphics Interface 1993*, 227–36.
- Rusinkiewicz, S. 1998. A new change of variables for efficient BRDF representation. In *Proceedings of the Eurographics Rendering Workshop*, 11–23.
- Rusinkiewicz, S., and M. Levoy. 2000. Qsplat: a multiresolution point rendering system for large meshes. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, 343–52.
- Sadeghi, I., O. Bisker, J. De Deken, and H. W. Jensen. 2013. A practical microcylinder appearance model for cloth rendering. *ACM Transactions on Graphics* 32(2), 14:1–14:12.
- Sadeghi, I., H. Pritchett, H. W. Jensen, and R. Tamstorf. 2010. An artist friendly hair shading system. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010)* 29(4), 56:1–56:10.
- Salesin, D., J. Stolfi, and L. Guibas. 1989. Epsilon geometry: building robust algorithms from imprecise computations. In *Proceedings of the Fifth Annual Symposium on Computational Geometry (SCG '89)*, 208–17.

- Saito, T., and T. Takahashi. 1990. Comprehensible rendering of 3-D shapes. In *Computer Graphics (Proceedings of SIGGRAPH '90)*, Volume 24, 197–206.
- Sattler, M., R. Sarlette, and R. Klein. 2003. Efficient and realistic visualization of cloth. *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering*, 167–78.
- Schaufler, G., and H. W. Jensen. 2000. Ray tracing point sampled geometry. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, 319–28.
- Schilling, A. 1997. Toward real-time photorealistic rendering: challenges and solutions. In *1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 7–16.
- Schwarzaupt, J., H. W. Jensen, and W. Jarosz. 2012. Practical Hessian-based error control for irradiance caching. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 31(6), 193:1–193:10.
- Schilling, A. 2001. Antialiasing of environment maps. *Computer Graphics Forum* 20(1), 5–11.
- Schlick, C. 1993. A customizable reflectance model for everyday rendering. In *Fourth Eurographics Workshop on Rendering*, Paris, France, 73–84.
- Schneider, P. J., and D. H. Eberly. 2003. *Geometric Tools for Computer Graphics*. San Francisco: Morgan Kaufmann.
- Schröder, K., R. Klein, and A. Zinke. 2011. A volumetric approach to predictive rendering of fabrics. *Computer Graphics Forum (Proceedings of the 2011 Eurographics Symposium on Rendering)* 30(4), 1277–86.
- Schuster, A. 1905. Radiation through a foggy atmosphere. *Astrophysical Journal* 21(1), 1–22.
- Schwarz, K. 2011. Darts, dice, and coins: sampling from a discrete distribution. <http://www.keithschwarz.com/darts-dice-coins/>.
- Schwarzschild, K. 1906. On the equilibrium of the sun's atmosphere (Nachrichten von der Königlichen Gesellschaft der Wissenschaften zu Göttingen). *Göttinger Nachrichten* 195, 41–53.
- Segovia, B., and M. Ernst. 2010. Memory efficient ray tracing with hierarchical mesh quantization. In *Proceedings of Graphics Interface 2010*, 153–60.
- Selle, A. 2015. Walt Disney Animation Studio's Hyperion renderer: engineering global illumination coherence at a production scale. *High Performance Graphics 2015 Hot3D Session*.
- Sen, P., B. Chen, G. Garg, S. Marschner, H. Mark, M. Horowitz, and H. P. A. Lensch. 2005. Dual photography. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)* 24(3), 745–55.
- Sen, P., and S. Darabi. 2011. Compressive rendering: a rendering application of compressed sensing. *IEEE Transactions on Visualization and Computer Graphics* 17(4), 487–99.
- Shade, J., S. J. Gortler, L. W. He, and R. Szeliski. 1998. Layered depth images. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, 231–42.
- Shevtsov, M., A. Soupikov, and A. Kapustin. 2007a. Ray-triangle intersection algorithm for modern CPU architectures. In *Proceedings of GraphiCon 2007*, 33–39.
- Shevtsov, M., A. Soupikov, and A. Kapustin. 2007b. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. In *Computer Graphics Forum (Proceedings of Eurographics 2007)* 26(3), 395–404.
- Shinya, M. 1993. Spatial anti-aliasing for animation sequences with spatio-temporal filtering. In *Proceedings of SIGGRAPH '93*, Computer Graphics Proceedings, Annual Conference Series, 289–96.

- Shinya, M., T. Takahashi, and S. Naito. 1987. Principles and applications of pencil tracing. In *Computer Graphics (Proceedings of SIGGRAPH '87)*, Volume 21, 45–54.
- Shirley, P. 1990. Physically based lighting calculations for computer graphics. Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana–Champaign.
- Shirley, P. 1991. Discrepancy as a quality measure for sample distributions. *Eurographics '91*, 183–94.
- Shirley, P. 1992. Nonuniform random point sets via warping. In D. Kirk (Ed.), *Graphics Gems III*, 80–83. San Diego: Academic Press.
- Shirley, P. 2011. Improved code for concentric map. <http://psgraphics.blogspot.com/2011/01/improved-code-for-concentric-map.html>.
- Shirley, P., and K. Chiu. 1997. A low distortion map between disk and square. *Journal of Graphics Tools* 2(3), 45–52.
- Shirley, P., and R. K. Morley. 2003. *Realistic Ray Tracing*. Natick, Massachusetts: A. K. Peters.
- Shirley, P., B. Wade, P. Hubbard, D. Zareski, B. Walter, and D. P. Greenberg. 1995. Global illumination via density estimation. In *Eurographics Rendering Workshop 1995*, 219–31.
- Shirley, P., C. Y. Wang, and K. Zimmerman. 1996. Monte Carlo techniques for direct lighting calculations. *ACM Transactions on Graphics* 15(1), 1–36.
- Shoemake, K. 1985. Animating rotation with quaternion curves, *Computer Graphics (SIGGRAPH '85 Proceedings)*, Volume 19, 245–54.
- Shoemake, K. 1991. Quaternions and 4x4 matrices. In J. Arvo (Ed.), *Graphics Gems II*, 351–54. San Diego: Academic Press.
- Shoemake, K. 1994a. Polar matrix decomposition. In P. Heckbert (Ed.), *Graphics Gems IV*, 207–21. San Diego: Academic Press.
- Shoemake, K. 1994b. Euler angle conversion. In P. Heckbert (Ed.), *Graphics Gems IV*, 222–29. San Diego: Academic Press.
- Shoemake, K., and T. Duff. 1992. Matrix animation and polar decomposition. In *Proceedings of Graphics Interface 1992*, 258–64.
- Sillion, F., and C. Puech. 1994. *Radiosity and Global Illumination*. San Francisco: Morgan Kaufmann.
- Simonot, L. 2009. Photometric model of diffuse surfaces described as a distribution of interfaced Lambertian facets *Applied Optics* 48(30), 5793–801.
- Sims, K. 1991. Artificial evolution for computer graphics. In *Computer Graphics (Proceedings of SIGGRAPH '91)*, Volume 25, 319–28.
- Slusallek, P. 1996. Vision—an architecture for physically-based rendering. Ph.D. thesis, University of Erlangen.
- Slusallek, P., and H.-P. Seidel. 1995. Vision—an architecture for global illumination calculations. *IEEE Transactions on Visualization and Computer Graphics* 1(1), 77–96.
- Slusallek, P., and H.-P. Seidel. 1996. Towards an open rendering kernel for image synthesis. In *Eurographics Rendering Workshop 1996*, 51–60.
- Smith, A. R. 1984. Plants, fractals and formal languages. In *Computer Graphics (Proceedings of SIGGRAPH '84)*, Volume 18, 1–10.
- Smith, A. R. 1995. A pixel is not a little square, a pixel is not a little square, a pixel is not a little square! (and a voxel is not a little cube). *Microsoft Technical Memo 6*.

- Smith, B. 1967. Geometrical shadowing of a random rough surface. *IEEE Transactions on Antennas and Propagation* 15(5), 668–71.
- Smith, J. O. 2002. Digital audio resampling home page. www-ccrma.stanford.edu/~jos/resample/.
- Smith, W. 2007. *Modern Optical Engineering* (4th ed.). New York: McGraw-Hill Professional.
- Smits, B. 1999. An RGB-to-spectrum conversion for reflectances. *Journal of Graphics Tools* 4(4), 11–22.
- Smits, B., P. S. Shirley, and M. M. Stark. 2000. Direct ray tracing of displacement mapped triangles. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, 307–18.
- Snow, J. 2010. Terminators and Iron Men: image-based lighting and physical shading at ILM. *SIGGRAPH 2010 Course: Physically-Based Shading Models in Film and Game Production*.
- Snyder, J. M., and A. H. Barr. 1987. Ray tracing complex models containing surface tessellations. *Computer Graphics (SIGGRAPH '87 Proceedings)*, Volume 21, 119–28.
- Sobol', I. 1967. On the distribution of points in a cube and the approximate evaluation of integrals. *Zh. vychisl. Mat. mat. Fiz.* 7(4), 784–802.
- Soupikov, A., M. Shevtsov, and A. Kapustin. 2008. Improving kd-tree quality at a reasonable construction cost. In *IEEE Symposium on Interactive Ray Tracing*, 67–72.
- Spanier, J., and E. M. Gelbard. 1969. *Monte Carlo Principles and Neutron Transport Problems*. Reading, Massachusetts: Addison-Wesley.
- Spencer, B., and M. Jones. 2009a. Hierarchical photon mapping. *IEEE Transactions on Visualization and Computer Graphics* 15(1), 49–61.
- Spencer, B., and M. Jones. 2009b. Into the blue: better caustics through photon relaxation. *Computer Graphics Forum (Proceedings of Eurographics 2009)* 28(2), 319–28.
- Stam, J. 1995. Multiple scattering as a diffusion process. In *Rendering Techniques (Proceedings of the Eurographics Rendering Workshop)*, 41–50.
- Stam, J. 1998. Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. In *Proceedings of SIGGRAPH '98*, Computer Graphics Proceedings, Annual Conference Series, 395–404.
- Stam, J. 1999. Diffraction shaders. In *Proceedings of SIGGRAPH '99*, Computer Graphics Proceedings, Annual Conference Series, 101–10.
- Stam, J. 2001. An illumination model for a skin layer bounded by rough surfaces. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, 39–52.
- Stam, J., and C. Loop. 2003. Quad/triangle subdivision. *Computer Graphics Forum* 22(1), 79–85.
- Stark, M., J. Arvo, and B. Smits. 2005. Barycentric parameterizations for isotropic BRDFs. *IEEE Transactions on Visualization and Computer Graphics* 11(2), 126–38.
- Steigleder, M., and M. McCool. 2003. Generalized stratified sampling using the Hilbert curve. *Journal of Graphics Tools* 8(3), 41–47.
- Steinert, B., H. Dammertz., J. Hanika, and H. P. A. Lensch. 2011. General spectral camera lens simulation. *Computer Graphics Forum* 30(6), 1643–54.
- Stephenson, I. 2006. Improving motion blur: shutter efficiency and temporal sampling. *Journal of Graphics Tools* 12(1), 9–15.
- Stich, M., H. Friedrich, and A. Dietrich. 2009. Spatial splits in bounding volume hierarchies. In *Proceedings of High Performance Graphics 2009*, 7–14.

- Stolfi, J. 1991. *Oriented Projective Geometry*. San Diego: Academic Press.
- Stürzlinger, W. 1998. Ray tracing triangular trimmed free-form surfaces. *IEEE Transactions on Visualization and Computer Graphics* 4(3), 202–14.
- Subr, K., and J. Arvo. 2007a. Statistical hypothesis testing for assessing Monte Carlo estimators: applications to image synthesis. In *Pacific Graphics '97*, 106–15.
- Subr, K., and J. Arvo. 2007b. Steerable importance sampling. *IEEE Symposium on Interactive Ray Tracing*, 133–40.
- Subr, K., and J. Kautz. 2013. Fourier analysis of stochastic sampling strategies for assessing bias and variance in integration. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2013)* 32(4), 128:1–128:12.
- Subr, K., D. Nowrouzezahrai, W. Jarosz, J. Kautz, and K. Mitchell. 2014. Error analysis of estimators that use combinations of stochastic sampling strategies for direct illumination. *Computer Graphics Forum (Proceedings of the 2014 Eurographics Symposium on Rendering)* 33(4), 93–102.
- Suffern, K. 2007. *Ray Tracing from the Ground Up*. Natick, Massachusetts: A. K. Peters.
- Sun, B., R. Ramamoorthi, S. Narasimhan, and S. Nayar. 2005. A practical analytic single scattering model for real time rendering. *ACM Transactions on Graphics* 24(3), 1040–49.
- Sun, B., K. Sunkavalli, R. Ramamoorthi, P. Belhumeur, and S. Nayar. 2007. Time-varying BRDFs. *IEEE Transactions on Visualization and Computer Graphics* 13(3), 595–609.
- Sun, Y., F. D. Fracchia, M. S. Drew, and T. W. Calvert. 2001. A spectrally based framework for realistic image synthesis. *The Visual Computer* 17(7), 429–44.
- Sung, K., and P. Shirley. 1992. Ray tracing with the BSP tree. In D. Kirk (Ed.), *Graphics Gems III*, 271–274. San Diego: Academic Press.
- Sung, K., J. Craighead, C. Wang, S. Bakshi, A. Pearce, and A. Woo. 1998. Design and implementation of the Maya renderer. In *Pacific Graphics '98*.
- Sutherland, I. E. 1963. Sketchpad—a man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference (AFIPS)*, 328–46.
- Suykens, F., and Y. Willems. 2001. Path differentials and applications. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, 257–68.
- Szirmay-Kalos, L., and G. Márton. 1998. Worst-case versus average case complexity of ray-shooting. *Computing* 61(2), 103–31.
- Szirmay-Kalos, L., B. Toth, and M. Magdic. 2011. Path sampling in high resolution inhomogeneous participating media. *Computer Graphics Forum* 30(1), 85–97.
- Szirmay-Kalos, L., M. Sbert, and T. Umenhoffer. 2005. Real-time multiple scattering in participating media with illumination networks. *Rendering Techniques 2005: 16th Eurographics Workshop on Rendering*, 277–82.
- Tabellion, E., and A. Lamorlette. 2004. An approximate global illumination system for computer generated films. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004)* 23(3), 469–76.
- Talbot, J., D. Cline, and P. Egbert. 2005. Importance resampling for global illumination. *Rendering Techniques 2005: 16th Eurographics Workshop on Rendering*, 139–46.
- Tannenbaum, D. C., P. Tannenbaum, and M. J. Wozny. 1994. Polarization and birefringency considerations in rendering. In *Proceedings of SIGGRAPH '94*, Computer Graphics Proceedings, Annual Conference Series, 221–22.

- Theußl, T., H. Hauser, and E. Gröller. 2000. Mastering windows: improving reconstruction. In *Proceedings of the 2000 IEEE Symposium on Volume Visualization*, 101–8. New York: ACM Press.
- Tong, X., J. Wang, S. Lin, B. Guo, and H. Y. Shum. 2005. Modeling and rendering of quasi-homogeneous materials. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)* 24(3), 1054–61.
- Torrance, K. E., and E. M. Sparrow. 1967. Theory for off-specular reflection from roughened surfaces. *Journal of the Optical Society of America* 57(9), 1105–14.
- Tregenza, P. R. 1983. The Monte Carlo method in lighting calculations. *Lighting Research and Technology* 15(4), 163–70.
- Trowbridge, S., and K. P. Reitz. 1975. Average irregularity representation of a rough ray reflection. *Journal of the Optical Society of America* 65(5), 531–36.
- Trumbore, B., W. Lytle, and D. P. Greenberg. 1993. A testbed for image synthesis. In *Developing Large-Scale Graphics Software Toolkits*, SIGGRAPH '93 Course Notes, Volume 3, 4-7–4-19.
- Truong, D. N., F. Bodin, and A. Seznec. 1998. Improving cache behavior of dynamically allocated data structures. In *IEEE PACT*, 322–29.
- Tsakok, J. 2009. Faster incoherent rays: multi-BVH ray stream tracing. In *Proceedings of High Performance Graphics 2009*, 151–58.
- Tumblin, J., and H. E. Rushmeier. 1993. Tone reproduction for realistic images. *IEEE Computer Graphics and Applications* 13(6), 42–48.
- Turk, G. 1991. Generating textures for arbitrary surfaces using reaction-diffusion. In *Computer Graphics (Proceedings of SIGGRAPH '91)*, Volume 25, 289–98.
- Turkowski, K. 1990a. Filters for common resampling tasks. In A. S. Glassner (Ed.), *Graphics Gems I*, 147–65. San Diego: Academic Press.
- Turkowski, K. 1990b. Properties of surface-normal transformations. In A. S. Glassner (Ed.), *Graphics Gems I*, 539–47. San Diego: Academic Press.
- Turkowski, K. 1993. The differential geometry of texture-mapping and shading. Technical Note, Advanced Technology Group, Apple Computer.
- Twomey, S., H. Jacobowitz, and H. B. Howell. 1966. Matrix methods for multiple-scattering problems. *Journal of the Atmospheric Sciences* 32, 289–96.
- Unger, J., A. Wenger, T. Hawkins, A. Gardner, and P. Debevec. 2003. Capturing and rendering with incident light fields. In *Proceedings of the Eurographics Rendering Workshop 2003*, 141–49.
- Unger, J., S. Gustavson, P. Larsson, and A. Ynnerman. 2008. Free form incident light fields. *Computer Graphics Forum (Proceedings of the 2008 Eurographics Symposium on Rendering)* 27(4), 1293–1301.
- Unser, M. 2000. Sampling—50 years after Shannon. In *Proceedings of the IEEE* 88(4), 569–87.
- Upstill, S. 1989. *The RenderMan Companion*. Reading, Massachusetts: Addison-Wesley.
- Ureña, C., M. Fajardo and A. King. 2013. An area-preserving parametrization for spherical rectangles. *Computer Graphics Forum (Proceedings of the 2013 Eurographics Symposium on Rendering)* 32(4), 59–66.
- van de Hulst, H. C. 1980. *Multiple Light Scattering*. New York: Academic Press.
- van de Hulst, H. C. 1981. *Light Scattering by Small Particles*. New York: Dover Publications. Originally published by John Wiley & Sons, 1957.

- Van Horn, B., and G. Turk. 2008. Antialiasing procedural shaders with reduction maps. *IEEE Transactions on Visualization and Computer Graphics* 14(3), 539–50.
- van Swaaij, M. 2006. Ray-tracing fur for Ice Age: The Melt Down. *ACM SIGGRAPH 2006 Sketches*.
- van Wijk, J. J. 1991. Spot noise-texture synthesis for data visualization. In *Computer Graphics (Proceedings of SIGGRAPH '91)*, Volume 25, 309–18.
- Veach, E. 1996. Non-symmetric scattering in light transport algorithms. In X. Pueyo and P. Schröder (Eds.), *Eurographics Rendering Workshop 1996*. Wien: Springer.
- Veach, E. 1997. Robust Monte Carlo methods for light transport simulation. Ph.D. thesis, Stanford University.
- Veach, E., and L. Guibas. 1994. Bidirectional estimators for light transport. In *Fifth Eurographics Workshop on Rendering*, Darmstadt, Germany, 147–62.
- Veach, E., and L. J. Guibas. 1995. Optimally combining sampling techniques for Monte Carlo rendering. In *Computer Graphics (SIGGRAPH '95 Proceedings)*, 419–28.
- Veach, E., and L. J. Guibas. 1997. Metropolis light transport. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, 65–76.
- Velázquez-Armendáriz, E., Z. Dong, B. Walter, D. P. Greenberg. 2015. Complex luminaires: illumination and appearance rendering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2015)* 34(3), 26:1–26:15.
- Verbeck, C. P., and D. P. Greenberg. 1984. A comprehensive light source description for computer graphics. *IEEE Computer Graphics and Applications* 4(7), 66–75.
- Villemin, R., and C. Hery. 2013. Practical illumination from flames. *Journal of Computer Graphics Techniques (JCGT)* 2(2), 142–55.
- Vinkler, M., V. Havran, and J. Sochora. 2012. Visibility driven BVH build up algorithm for ray tracing. *Computers & Graphics* 36(4), 283–96.
- Vorba, J., and O. Karlík, M. Šik, T. Ritschel, and J. Křivánek. 2014. On-line learning of parametric mixture models for light transport simulation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014)* 33(4), 101:1–101:11.
- Wächter, C. A. 2008. Quasi Monte Carlo light transport simulation by efficient ray tracing. Ph.D. thesis, University of Ulm.
- Wächter, C. A., and A. Keller. 2006. Instant ray tracing: the bounding interval hierarchy. In *Rendering Techniques 2006: 17th Eurographics Workshop on Rendering*, 139–49.
- Wald, I. 1999. Photorealistic rendering using the PhotonMap. Diploma thesis, Universität Kaiserslautern.
- Wald, I. 2007. On fast construction of SAH-based bounding volume hierarchies. In *IEEE Symposium on Interactive Ray Tracing*, 33–40.
- Wald, I. 2012. Fast construction of SAH BVHs on the Intel Many Integrated Core (MIC) architecture. *IEEE Transactions on Visualization and Computer Graphics* 18(1), 47–57.
- Wald, I., C. Benthin, and S. Boulos. 2008. Getting rid of packets—efficient SIMD single-ray traversal using multibranching BVHs. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing 2008*, 49–57.
- Wald, I., C. Benthin, and P. Slusallek. 2003. Interactive global illumination in complex and highly occluded environments. In *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering*, 74–81.

- Wald, I., and V. Havran. 2006. On building fast kd-trees for ray tracing and on doing that in $O(n \log n)$. In *IEEE Symposium on Interactive Ray Tracing*, 61–69.
- Wald, I., P. Slusallek, and C. Benthin. 2001b. Interactive distributed ray tracing of highly complex models. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, 277–88.
- Wald, I., T. Kollig, C. Benthin, A. Keller, and P. Slusallek. 2002. Interactive global illumination using fast ray tracing. In *Rendering Techniques 2002: 13th Eurographics Workshop on Rendering*, 15–24.
- Wald, I., S. Boulos, and P. Shirley. 2007a. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics* 26(1).
- Wald, I., W. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. Parker, and P. Shirley. 2007b. State of the art in ray tracing animated scenes. In *Eurographics 2007 State of the Art Reports*.
- Wald, I., P. Slusallek, C. Benthin, and M. Wagner. 2001a. Interactive rendering with coherent ray tracing. *Computer Graphics Forum* 20(3), 153–64.
- Wald, I., S. Woop, C. Benthin, G. S. Johnson, and M. Ernst. 2014. Embree: a kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014)* 33(4), 143:1–143:8.
- Walker, A. J. 1974. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electronics Letters* 10(8): 127–28.
- Walker, A. J. 1977. An efficient method for generating discrete random variables with general distributions. *ACM Transactions on Mathematical Software* 3(3), 253–56.
- Wallis, B. 1990. Forms, vectors, and transforms. In A. S. Glassner (Ed.), *Graphics Gems I*, 533–38. San Diego: Academic Press.
- Walter, B., A. Arbree, K. Bala, D. Greenberg. 2006. Multidimensional lightcuts. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006)* 25(3), 1081–88.
- Walter, B., S. Fernandez, A. Arbree, K. Bala, M. Donikian, D. Greenberg. 2005. Lightcuts: a scalable approach to illumination. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)* 24(3), 1098–107.
- Walter, B., P. M. Hubbard, P. Shirley, and D. F. Greenberg. 1997. Global illumination using local linear density estimation. *ACM Transactions on Graphics* 16(3), 217–59.
- Walter, B., P. Khungurn, and K. Bala. 2012. Bidirectional lightcuts. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2012)* 31(4), 59:1–59:11.
- Walter, B., S. Marschner, H. Li, and K. Torrance. 2007. Microfacet models for refraction through rough surfaces. In *Rendering Techniques 2007 (Proc. Eurographics Symposium on Rendering)*, 195–206.
- Walter, B., K. Bala, M. Kilkarni, and K. Pingali. 2008. Fast agglomerative clustering for rendering. In *IEEE Symposium on Interactive Ray Tracing*, 81–86.
- Walter, B., S. Zhao, N. Holzschuch, and K. Bala. 2009. Single scattering in refractive media with triangle mesh boundaries. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2009)* 28(3), 92:1–92:8.
- Wandell, B. 1995. *Foundations of Vision*. Sunderland, Massachusetts: Sinauer Associates.
- Wang, J., S. Zhao, X. Tong, J. Snyder, and B. Guo. 2008a. Modeling anisotropic surface reflectance with example-based microfacet synthesis. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008)* 27(3), 41:1–41:9.

- Wang, J., S. Zhao, X. Tong, S. Lin, Z. Lin, Y. Dong, B. Guo, and H. Y. Shum. 2008b. Modeling and rendering of heterogeneous translucent materials using the diffusion equation. *ACM Transactions on Graphics* 27(1), 9:1–9:18.
- Wang, R., and O. Åkerlund. 2009. Bidirectional importance sampling for unstructured illumination. *Computer Graphics Forum (Proceedings of Eurographics 2009)* 28(2), 269–78.
- Wang, X. C., J. Maillet, E. L. Fiume, V. Ng-Thow-Hing, A. Woo, and S. Bakshi. 2000. Feature-based displacement mapping. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, 257–68.
- Ward, G. 1991. Adaptive shadow testing for ray tracing. In *Second Eurographics Workshop on Rendering*.
- Ward, G. 1992. Real pixels. In J. Arvo (Ed.), *Graphics Gems IV*, 80–83. San Diego: Academic Press.
- Ward, G. J. 1994. The Radiance lighting simulation and rendering system. In *Proceedings of SIGGRAPH '94*, 459–72.
- Ward, G., and E. Eydelberg-Vilesin. 2002. Picture perfect RGB rendering using spectral prefiltering and sharp color primaries. In *Proceedings of 13th Eurographics Workshop on Rendering*, Pisa, Italy, 117–24.
- Ward, G. J., F. M. Rubinstein, and R. D. Clear. 1988. A ray tracing solution for diffuse interreflection. *Computer Graphics (SIGGRAPH '88 Proceedings)*, Volume 22, 85–92.
- Ward, K., F. Bertails, T.-Y. Kim, S. R. Marschner, M.-P. Cani, and M. Lin. 2007. A survey on hair modeling: styling, simulation, and rendering. *IEEE Transactions on Visualization and Computer Graphics* 13(2), 213–34.
- Warn, D. R. 1983. Lighting controls for synthetic images. In *Computer Graphics (Proceedings of SIGGRAPH 83)*, Volume 17, 13–21.
- Warren, H. 2006. *Hacker's Delight*. Reading, Massachusetts: Addison-Wesley.
- Warren, J. 2002. *Subdivision Methods for Geometric Design: A Constructive Approach*. San Francisco: Morgan Kaufmann.
- Weghorst, H., G. Hooper, and D. P. Greenberg. 1984. Improved computational methods for ray tracing. *ACM Transactions on Graphics* 3(1), 52–69.
- Wei, L.-Y. 2008. Parallel Poisson disk sampling. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008)* 27(3), 20:1–20:10.
- Wei, L.-Y., S. Lefebvre, V. Kwatra, and G. Turk. 2009. State of the art in example-based texture synthesis. In *Eurographics 2009, State of the Art Report*.
- Weistroffer, R. P., K. Walcott, G. Humphreys, and J. Lawrence. 2007. Efficient basis decomposition for scattered reflectance data. *Eurographics Symposium on Rendering*, 207–18.
- Westin, S., J. Arvo, and K. Torrance. 1992. Predicting reflectance functions from complex surfaces. *Computer Graphics* 26(2), 255–64.
- Weyrich, T., P. Peers, W. Matusik, and S. Rusinkiewicz. 2009. Fabricating microgeometry for custom surface reflectance *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008)* 28(3), 32:1–32:6.
- Whitted, T. 1980. An improved illumination model for shaded display. *Communications of the ACM* 23(6), 343–49.

- Weidlich, A., A. Wilkie. 2007. Arbitrarily layered micro-facet surfaces. In *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia (GRAPHITE '07)*, 171–78.
- Wilkie, A., S. Nawaz, M. Droske, A. Weidlich, and J. Hanika. 2014. Hero wavelength spectral sampling. *Computer Graphics Forum (Proceedings of the 2014 Eurographics Symposium on Rendering)* 33(4), 123–31.
- Wilkie, A., and A. Weidlich. 2009. A robust illumination estimate for chromatic adaptation in rendered images. *Computer Graphics Forum (Proceedings of the 2009 Eurographics Symposium on Rendering)* 28(4), 1101–09.
- Wilkie, A., and A. Weidlich. 2011. A physically plausible model for light emission from glowing solid objects. *Computer Graphics Forum (Proceedings of the 2011 Eurographics Symposium on Rendering)* 30(4), 1269–76.
- Wilkie, A., A. Weidlich, C. Larboulette, and W. Purgathofer. 2006. A reflectance model for diffuse fluorescent surfaces. In *Proceedings of GRAPHITE*, 321–31.
- Wilkinson, J. H. 1994. *Rounding Errors in Algebraic Processes*. New York: Dover Publications, Inc. Originally published by Prentice-Hall Inc., 1963.
- Williams, L. 1978. Casting curved shadows on curved surfaces. In *Computer Graphics (Proceedings of SIGGRAPH '78)*, Volume 12, 270–74.
- Williams, L. 1983. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, Volume 17, 1–11.
- Williams, A., S. Barrus, R. K. Morley, and P. Shirley. 2005. An efficient and robust ray–box intersection algorithm. *Journal of Graphics, GPU, and Game Tools* 10(4), 49–54.
- Wilson, P. R., M. S. Johnstone, M. Neely, and D. Boles. 1995. Dynamic storage allocation: a survey and critical review. In *Proceedings International Workshop on Memory Management*, Kinross, Scotland.
- Witkin, A., and M. Kass. 1991. Reaction-diffusion textures. In *Computer Graphics (Proceedings of SIGGRAPH '91)*, Volume 25, 299–308.
- Wolff, L. B., and D. J. Kurlander. 1990. Ray tracing with polarization parameters. *IEEE Computer Graphics and Applications* 10(6), 44–55.
- Woo, A., and J. Amanatides. 1990. Voxel occlusion testing: a shadow determination accelerator for ray tracing. In *Proceedings of Graphics Interface 1990*, 213–20.
- Woo, A., A. Pearce, and M. Ouellette. 1996. It's really not a rendering bug, you see *IEEE Computer Graphics and Applications* 16(5), 21–25.
- Woodcock, E., T. Murphy, P. Hemmings, and T. Longworth. 1965. Techniques used in the GEM code for Monte Carlo neutronics calculations in reactors and other systems of complex geometry. *Proc. Conference on the Application of Computing Methods to Reactor Problems, ANL-7050*, 557–79.
- Woop, S., C. Benthin, and I. Wald. 2013. Watertight ray/triangle intersection. *Journal of Computer Graphics Techniques (JCGT)* 2(1), 65–82.
- Woop, S., C. Benthin, I. Wald, G. S. Johnson, and E. Tabellion. 2014. Exploiting local orientation similarity for efficient ray traversal of hair and fur. In *Proceedings of High Performance Graphics 2014*, 41–49.
- Woop, S., G. Marmitt, and P. Slusallek. 2006. B-kd trees for hardware accelerated ray tracing of dynamic scenes. In *Graphics Hardware 2006: Eurographics Symposium Proceedings*, Vienna, Austria, 67–76.

- Woop, S., J. Schmittler, and P. Slusallek. 2005. RPU: a programmable ray processing unit for realtime ray tracing. In *ACM SIGGRAPH 2005 Papers*, 434–44.
- Worley, S. P. 1996. A cellular texture basis function. In *Proceedings of SIGGRAPH '96*, Computer Graphics Proceedings, Annual Conference Series, New Orleans, Louisiana, 291–94.
- Wrenninge, M. 2012. *Production Volume Rendering: Design and Implementation*. Boca Raton, Florida: A. K. Peters/CRC Press.
- Wrenninge, M. 2015. Field3D. <http://magnuswrenninge.com/field3d>.
- Wrenninge, M., C. Kulla, and V. Lundqvist. 2013. Oz: the great and volumetric. In *ACM SIGGRAPH 2013 Talks*, 46:1–46:1.
- Wu, H., J. Dorsey, and H. Rushmeier. 2011. Physically-based interactive bi-scale material design. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2011)* 30(6), 145:1–145:10.
- Wyman, D., M. Patterson, and B. Wilson. 1989. Similarity relations for anisotropic scattering in Monte Carlo simulations of deeply penetrating neutral particles. *Journal of Computational Physics* 81, 137–50.
- Wyyvill, B., and G. Wyyvill. 1989. Field functions for implicit surfaces. *The Visual Computer* 5(1/2), 75–82.
- Yan, L.-Q., M. Hašan, W. Jakob, J. Lawrence, S. Marschner, and R. Ramamoorthi. 2014. Rendering glints on high-resolution normal-mapped specular surfaces. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014)* 33(4), 116:1–116:9.
- Yanovitskij, E. G. 1997. *Light Scattering in Inhomogeneous Atmospheres*. Berlin: Springer-Verlag.
- Yellot, J. I. 1983. Spectral consequences of photoreceptor sampling in the Rhesus retina. *Science* 221, 382–85.
- Yoon, S.-E., S. Curtis, and D. Manocha. 2007. Ray tracing dynamic scenes using selective restructuring. In *Proceedings of the Eurographics Symposium on Rendering*, 73–84.
- Yoon, S.-E., and P. Lindstrom. 2006. Mesh layouts for block-based caches. *IEEE Transactions on Visualization and Computer Graphics*, 12(5), 1213–20.
- Yoon, S.-E., and D. Manocha. 2006. Cache-efficient layouts of bounding volume hierarchies. In *Computer Graphics Forum: Proceedings of Eurographics 2006* 25(3), 507–16.
- Yoon, S.-E., P. Lindstrom, V. Pascucci, and D. Manocha. 2005. Cache-oblivious mesh layouts. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)* 24(3), 886–93.
- Yoon, S.-E., C. Lauterbach, and D. Manocha. 2006. R-LODs: fast LOD-based ray tracing of massive models. *The Visual Computer* 22(9–11), 772–84.
- Yue, Y., K. Iwasaki, B.-Y. Chen, Y. Dobashi, and T. Nishita. 2010. Unbiased, adaptive stochastic sampling for rendering inhomogeneous participating media. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2010)* 29(5), 177:1–177:7.
- Yue, Y., K. Iwasaki, B.-Y. Chen, Y. Dobashi, and T. Nishita. 2011. Toward optimal space partitioning for unbiased, adaptive free path sampling of inhomogeneous participating media. *Computer Graphics Forum* 30(7), 1911–19.
- Zachmann, G. 2002. Minimal hierarchical collision detection. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, 121–28.
- Zhao, S., W. Jakob, S. Marschner, and K. Bala. 2011. Building volumetric appearance models of fabric using micro CT imaging. *ACM Transactions on Graphics* 30(4), 44:1–44:10.

- Zhao, S., W. Jakob, S. Marschner, and K. Bala. 2012. Structure-aware synthesis for predictive woven fabric appearance. *ACM Transactions on Graphics* 31(4), 75:1–75:10.
- Zhao, S., R. Ramamoorthi, and K. Bala. 2014. High-order similarity relations in radiative transfer. *ACM Transactions on Graphics* 33(4), 104:1–104:12.
- Zhou, K., Q. Hou, R. Wang, and B. Guo. 2008. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2008)* 27(5), 126:1–126:11.
- Zickler, T., S. Enrique, R. Ramamoorthi, and P. Belhumeur. 2005. Reflectance sharing: image-based rendering from a sparse set of images. *Rendering Techniques 2005 (Proceedings of the Eurographics Symposium on Rendering)*, 253–65.
- Zimmerman, K. 1995. Direct lighting models for ray tracing with cylindrical lamps. In *Graphics Gems V*, 285–89. San Diego: Academic Press.
- Zinke, A., C. Yuksel, A. Weber, and J. Keyser. 2008. Dual scattering approximation for fast multiple scattering in hair. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008)* 27(3), 32:1–32:10.
- Zorin, D., P. Schröder, T. DeRose, L. Kobbelt, A. Levin, and W. Sweldens. 2000. *Subdivision for Modeling and Animation*. SIGGRAPH 2000 Course Notes.
- Zsolnai, K., and L. Szirmay-Kalos. 2013. Automatic parameter control for Metropolis light transport. *Eurographics 2013 Short Paper*.
- Zuniga, M., and J. Uhlmann. 2006. Ray queries with wide object isolation and the S-tree. *Journal of Graphics, GPU, and Game Tools* 11(3), 27–45.
- Zwicker, M., W. Jarosz, J. Lehtinen, B. Moon, R. Ramamoorthi, F. Rousselle, P. Sen, C. Soler, and S.-E. Yoon. 2015. Recent advances in adaptive sampling and reconstruction for Monte Carlo rendering. *Computer Graphics Forum (Proceedings of Eurographics 2015)* 34(2), 667–81.

This page intentionally left blank

Subject Index

Numbers

(0,2)-sequences, 454–465
2D sample generation, 455
defined, 455
elementary intervals, 455
illustrated, 456
random scrambling, 458, 459
sample generation, 459
sequence length, 455
stratification, 455
unscrambled, 459
use of, 465
 2×2 linear systems, 1080
2D (u, v) mapping, 610–611
3D grids, 690–691
defined, 690
implementations, 690
local density computation, 691
voxel coordinates/offsets computation, 691
3D mapping, 613–614
3D viewing problem, 358
 4×4 matrices, 1080–1081

A

Absolute error, 215
Absorption, 672–674
cross section, 673
defined, 672
effect, 674–675
illustrated, 673, 674
radiance and, 674
total fraction of light, 674
Absorption coefficient
of conductors, 520
of volumes, 673
Abstract base classes, 15, 43
Abstraction
efficiency versus, 46
radiative transfer, 334
of shapes, 123
Acceleration structures
bounding volume hierarchies (BVHs), 255–284
defined, 247
goal, 254
grid accelerators, 308

kd-tree accelerator, 284–302
in ray-object intersection, 8
specialized, 1052
Acceptance probability, 763
Across tangents, 205
Adaptive sampling, 412, 497, 498, 504, 880
defined, 412
See also Sampling
Adjoint BSDF, 960
Affine space, 57
Affine transformations, 104
Aggregates, 254–255
aliases, 409
intersections, 255
Albedo, 676
in participating media, 676
reduced, 918
of skim milk, 904
Aliasing, 409–410
checkerboard textures, 643
defined, 402
illustrated, 410
from point sampling, 411
shading, 415
sources, 414–415
specular highlight, 593
texture, 597, 598
visual impact, 597
Alpha masks, 165
Amdahl’s law, 39
Angle measure conversion, 1063
Angle rotation, 89, 90
Animated shapes, 1123–1125, 1127
Animation
keyframe matrix, 97
primitives, 251–254
transformation, 97–99
Anisotropic surfaces, 509
Antialiasing
adaptive sampling, 412
checkerboard textures, 643
fractional Brownian motion, 656
mix textures and, 616
nonuniform sampling, 411
polka dots, 654
prefiltering, 412–413
procedural texturing, 640, 647
ray differentials and, 38, 74–75
scale texture and, 615
techniques, 410–413
texture, 599–604
Application programming interfaces (APIs)
cleanup, 1110, 1114
function definitions, 1113, 1115, 1116, 1118, 1119, 1121, 1123, 1127, 1128, 1130
initialization, 1110, 1114, 1118
local classes, 1112, 1114, 1118
macros, 1111, 1112, 1114
rendering option functions, 1115
states, 1110
static data, 1110, 1111, 1113, 1114, 1119
Area
differential, 345, 347
integrals over, 347–348
Area lights, 733–737
creation, 1124
in current graphics state, 1125
defined, 733–737
diffuse, 736, 957
emitted power, 736
illumination illustration, 735
incident illumination, 737
infinite, 737–714, 845–851, 959–960
interface, 734
sampling, 442, 464–465, 836–838
sampling methods, 845
See also Light sources; Light(s)
Area product measure, 1000
Arena-based allocation, 1073–1076
advantages, 1073
defined, 1073
memory areas, 1073–1076
See also Memory allocation; Memory management
Arithmetic logic units (ALUs), 1055
Ashikhmin–Shirley model, 550–551
Asymmetry parameter, 681, 683
Atomic operations, 41–42, 1084–1087
addition of floating-point values, 1085–1087
compare and swap, 1085
implementation of, 1086

Atomic operations (*continued*)

- lock-free algorithm, 1085
- mutexes versus, 42

Atomics. *See* Atomic operations

Attenuation

- coefficient, 676
- of light, 13
- reduced coefficient, 918

Attribute stacks, 1118

Axis-aligned bounding boxes (AABBs)

- defined, 76–77
- illustrated, 77

B

Back-scattering, 682

Backward error analysis, 216

Balance heuristic, 798, 1012, 1014

Band-limited functions, 406, 496, 604

Barrel distortion, 355

Barycentric coordinates, 162

- computation, 162–163
- rounding error, 226

Basis vectors, 57–58, 90

BDPT. *See* Bidirectional path tracing

Beam transmittance, 677–678

- defined, 677
- illustrated, 677, 678
- multiplicative property, 678
- properties, 676–678

Beckmann–Spizzichino microfacet distribution function, 538, 808–809

Bézier basis, 167

Bézier curve, 167

- convex hull property, 171

Bézier splines, 167

Bias

- photon mapping, 966
- start-up, 766
- in variance reduction, 793–794

Bidirectional path tracing, 990–1024

- camera subpath, 991

- common measure, 1000

- as generalization of standard path-tracing algorithm, 990

- illustrated, 993

- infinite area lights, 1019–1022

- light subpath, 991

- multiple importance sampling, 1012–1019

- subpath generation, 1003

- vertex abstraction, 995

See also Path tracing

Bidirectional reflectance distribution functions (BRDFs), 349–351

- adapter, 515

delta distribution, 523–524

energy conservation, 350

Fresnel incidence effects, 549–552

illustrated, 349

implementation, 513

interface, 512–514

Lambertian model, 532

memory management, 576

microfacet-based models, 533

public data, 513

reciprocity, 350

scaling adapter, 515

specular reflection, 516

utility functions, 519

Bidirectional scattering distribution

- functions (BSDFs), 571–576

- adjoint, 960

- BRDF reuse adapter, 515

- class representation, 571

- defined, 350

- destructors, 576

- direction conventions, 512

- implementations, 512, 573

- inline functions, 510

- interface, 512–514

- interface setting, 510

- memory management, 250, 576

- random variables, 832

- sample initialization, 832

- sample values, 830

- sampled directions, 832–833

- sampling, 832–835

- sampling distribution, 857

- sampling PDF computation, 834

- scaling adapter, 515

- utility functions, 519

Bidirectional scattering-surface

- reflectance distribution function (BSSRDF), 351–352, 692–695

- computation, 692

- defined, 350

- illustrated, 351

- scattering equation, 350

- separable, 693–696

- subsurface scattering materials, 700–702

Bidirectional transmittance distribution

- functions (BTDFs)

- defined, 350

- denotations, 350

- specular transmission, 526–531

Bilinear interpolation texture, 617–618

- defined, 617

See also Texture(s)

Binary space partitioning (BSP) trees

- defined, 284

kd-trees, 284–302

octrees, 284

Blackbody emitter, 709–712

Blinn's Law, 48

Blocked 2D arrays, 1076–1079

- arrangement in memory, 1076

- block size, 1076

- natural layout, 1077

Blossoming (splines), 172

Blurring, 479

Bootstrapping samples, 1038

Boundary conditions

- of diffusion dipole, 926

- of equation of transfer, 888

Boundary vertices, 189

- determining, 190

- limit rule, 203

- odd, 199

- one-ring, 196–197

- subdivision and, 194

- tangent computation, 204–205

- valence computation, 191–192

Bounding

- cylinder, 143

- disk, 146–150

- shape interface, 125

- sphere, 134

Bounding boxes

- axis-aligned (AABBs), 77

- BVH nodes, 258

- overlapping kd-tree nodes, 295

- points, 77, 78

- ray intersection, 126–129

- sphere, 81

- surface area computation, 80–81

- in texture space, 638

- three-dimensional, 75–81

- transforming, 95

Bounding spheres, 958, 959

Bounding volume, 124

Bounding volume hierarchies (BVHs), 255–284

- binary, 255

- bounds computation, 259

- characteristics, 255–256

- compact, for traversal, 280–282

- construction, 257–263

- defined, 255

- definition of, 256

- efficiency, 255

- illustrated example, 256

- interior nodes, 260, 282, 283

- leaf nodes, 258, 260, 283

- linear layout in memory, 280

- nodes, 258, 259, 281, 282

- nodes, bounding boxes, 283

- nodes, checking ray against, 283
 nodes, following ray through, 283
 partition axis, 260
 performance, 129
 primitive storage, 256
 surface area heuristic, 263–268
 traversal, 282–284
- Box filter, 475–477
 defined, 475
 graph, 476
 MIP map, 632
 reconstruction, 477
See also Filters; Reconstruction
- Bracketing interval, 825
- BRDF. *See* Bidirectional reflectance distribution function
- BSDF. *See* Bidirectional scattering distribution function
- BSSRDF. *See* Bidirectional scattering-surface reflectance distribution function
- BTDF. *See* Bidirectional transmittance distribution function
- Bugs, reporting, 47
- Bump mapping, 584–590
 appearance, 586
 application effect, 586, 587
 defined, 586
 displacement texture evaluation, 589
 implementations, 593
 offset position computation, 590
 offsets selection, 590
 in shading normals computation, 586
 in Sponza atrium model, 587
 support, 587
See also Materials
- Bumpy textures, 660–662
- BVHs. *See* Bounding volume hierarchies
- C**
- Cache coherency, 1082
- Cache-friendly memory usage, 1071–1073
- Caching
 geometry, 1052, 1053
 texture, 620–622
- Camera(s), 6–7
 class, 355–358
 coordinate spaces, 358
 defined, 5
 environment, 375–377
 implementation, 357
 interface, 357
 models, 356–399
 orthographic, 361–364
- perspective, 365–368
 physical lens system simulation, 377–397
 pinhole, 5, 6, 373
 projective, 358–375
 ray computation, 367
 real-world, 356
 sampling, 949–955
 simulator, 6–7
- Camera importance function, 949
- Camera measurement equation, 395–397
- Camera space
 defined, 358, 359
 origin, 359
 points in, 363, 366
 world space transformation to, 359
See also Coordinate spaces
- Camera subpath, 991
- Candelas, 342
- Canonical uniform random variables, 749
- Cardinal point, 386
- Cartesian coordinates, 773
- Catmull–Rom spline. *See* Spline
- Checkerboard textures
 aliasing, 643
 antialiasing, 643
 application illustration, 642
 check function frequency, 642–643
 closed-form box-filtered, 644
 defined, 642
 filtering, 644
 sampling method comparison, 436
 solid, 646–648
 step function integration, 645
 supersampling, 647
 texture coordinates, 609, 646
See also Texture(s)
- CIE standard illuminants, 712
- Circle of confusion, 369, 370
 illustrated, 372
- Clamping, 605, 656, 666
 function, 1062
- Classes
 abstract base, 15, 43
 geometric, 57
 relationships, 27, 33
 summarization, 16
 surface integration, 33
- Classical diffusion coefficient, 922
- Closed mesh, 185
- Code, this book
 bugs, 47
 optimization, 46
 pointer versus reference, 45
 using/understanding, 45–47
- Coherent shared memory, 39
- Color(s)
 out-of-gamut, 495
 RGB, 325–331
 tristimulus theory, 322
 XYZ, 322–325
- Color temperature, 711
- Communication with user, 1068–1070
 error reporting, 1068
 float file reader, 1070
 reporting progress, 1069
- Compare and swap, 1084
- Concentric mapping, 777, 778
- Conditional density function, 774
- Conductors, 517, 518
- Cones, 150–151
 implicit form, 152
 parametric form, 150–152
 partial derivatives, 150–152
See also Shapes
- Constant texture, 615
- Continuous coordinates, 415, 416
- Continuous random variables, 749–760
- Continuous transformations, 82
- Control mesh, 181–185
 consistently ordered, 186
 directed edge, 186
See also Subdivision mesh
- Control point, 174, 176
- Convex hull property, 171, 195
- Convolution
 computing, 406
 Fourier transform, 406
 operation, 406
 shah function, 407, 408
 in spatial domain, 407
- Coordinate spaces
 camera space, 358, 363, 366
 illustrated, 359
 normalized device coordinate (NDC) space, 359
 object space, 133, 358
 raster space, 359, 360
 world space, 58, 358
- Coordinate systems, 57–58
 defined, 57
 frame, 57, 58
 handedness, 58–59, 96
 left-handed, 58, 59
 reflection, 509
 right-handed, 58, 59
 shading, 509, 511
 from vectors, 67
 world space, 58
- Cosine-weighted hemisphere sampling, 779–780

CPUs
 cache coherence protocol, 1082
 packet tracing on, 1057
 SIMD width, 1059
 Critical angle, 520
 Cross product, 64–65
 defined, 64
 of perpendicular unit vectors, 65
 “up” vector, 92
 Cross-referencing, 3–4
 Cumulative distribution function (CDF)
 constructing, 753–754
 defined, 749
 discrete, 754
 integral, 754
 inversion, 755, 758
 monotonically increasing, 758
 normalization, 755
 PDF and, 755
 piecewise-constant 1D functions,
 757–760
 Current transformation matrices
 (CTMs), 1111–1119
 active, 1111
 for initialization, 1111
 named copies of, 1113
 pbrt maintenance, 1110
 selective modification, 1111
 use of, 1112
 Curse of dimensionality, 433
 Curves
 Bézier splines, 167, 171
 blossoming, 171–172
 intersections, 173–181
 types, 169
See also Shapes
 Cylinders, 142–146
 basic setting, 142
 bounding, 143
 coefficients computation, 144
 hit point computation, 144
 illustrated, 143
 implementation, 142
 implicit form, 144
 infinitely long, 144
 intersection, 144–146
 parametric form, 145
 partial derivatives, 145
 sampling, 839
 surface area, 146
See also Shapes
 Cylindrical mapping, 612

D

Data races
 coordination and, 39–42

defined, 39
See also Parallelization
 Decomposition
 function, 3
 matrix, 98
 polar, 104
 screen-space, 29
 transformation, 104
 Delta distributions, 405
 Dirac, 405, 815
 in integrand, 868–869
 light, 836
 Delta tracking, 895
 Denormalized floating-point number,
 210
 Depth of field, 368–375
 circle of confusion and, 369, 370
 defined, 374
 focal distance, 374
 illustrated, 370, 371
 lens aperture size, 370, 371
 orthographic camera, 363
 pinhole cameras and, 373
 projective cameras, 374
 Design
 abstraction versus efficiency, 46
 graphics API, 1104
 retrospective, 1051–1060
 triangles-only alternative, 1052–1053
 Detailed balance, 763
 Dielectric media, 518
 Differential area, 345, 347
 Differential geometry
 defined, 117
 illustrated, 117
 Differential irradiance, 9, 349
See also Irradiance
 Differential radiance, 349
See also Radiance
 Diffuse area lights, 957
 Diffuse reflection
 defined, 508
 illustrated, 509
 Oren–Nayar, 534–437
 Diffusion approximation, 917
 classical, 922
 non-classical, 024
 Diffusion equation, 916–922
 Dipole, 925
 Dipole approximation, 925
 basic setting for, 925
 boundary condition, 925
 Dipole diffusion solution, 925
 Dirac delta distribution, 405, 523, 815
 Direct lighting, 36, 851–856
 computation, 877
 estimating, 856–861
 integral, estimating, 856–861
 integrator, 851, 852, 854, 856
 multiple importance sampling, 858,
 859
 sampling, 856
 scene rendering, 851
See also Lighting
 Direct3D, 1103
 Directional lights, 731–733
 infinite light direction, 958
 light direction point offset, 958
 outgoing ray direction, 958
 sampling, 957–959
 Directions
 differential reflected, 608
 incident radiance, 349
 normalized, 34
 normals, 118
 ray, 363
 sampling, 834
 specular reflection, 525
 spherical coordinate representation,
 344, 773
 transformation into shading space,
 573
 transmitted, 36
 transmitted ray, 529
 vector representation, 59
 Discrepancy
 box, 419
 computing, 419–420
 defined, 417–421
 Halton sequence, 444
 star, 420
 van der Corput sequence, 443
 Discrete coordinates, 415
 Discrete random media, 943
 Disks, 146–150
 basic setting, 147
 bounding, 147–148
 generalized to annulus, 146
 hit point computation, 148
 illustrated, 147
 intersection, 148–149
 parameterization, inverting, 149
 parametric form, 146
 partial, 147
 partial derivatives, 149
 plane intersection computation, 148
 sampling, 776–779, 838
 surface area, 150
See also Shapes
 Dispersion, 516
 Displacement textures, 587, 589
 Display pixels, 402–403
 Distant lights, 731–733
 defined, 731

- power approximation, 733
See also Directional lights; Light sources; Light(s)
- Distributions
- conditional sampling, 785
 - delta, 405, 523, 868–869
 - exponential, 756
 - geometric, 8
 - half-angle, 812
 - joint, 773
 - light, 8–10, 15
 - marginal sampling, 774, 785
 - photon, 966
 - piecewise-constant 1D, 757–760
 - piecewise-constant 2D, 784–787
 - power, 755–756
 - radiometric, 8
 - spectral power (SPD), 313–318, 329, 333
 - stationary, 763
 - transforming between, 771–773
- Dot product, 63–65
- absolute value, 64
 - angle relationship, 63
 - computation between normals and vectors, 72
 - defined, 63
- E**
- Edge function, 161, 176
- Effective albedo, 698
- relation to single scattering albedo, 936
- Effective transport coefficient, 923
- Efficiency
- abstraction versus, 46
 - bias and, 793–794
 - BVHs, 255
 - estimator definition, 787
 - importance sampling and, 794–799
 - rejection method, 760
 - Russian roulette and, 787–789
 - sample placement and, 789–793
 - sampling reflection functions, 806–835
 - splitting and, 788–789
- Elementary intervals, 455
- Elliptically weighted average (EWA), 634–639
- coordinates conversion, 637
 - ellipse axes computation, 636
 - ellipse bounding box computation, 637
 - ellipse coefficients computation, 637
 - ellipse eccentricity, 635
 - filter illustration, 634
- filter table lookup, 639
- filter weights initialization, 639
- filtering algorithm, 604
- image pyramid, 636
- lookup detail, 636
- unified resampling filter, 635
- Emission, 672, 674–676
- defined, 672
 - differential equation, 675
 - illustrated, 675
- Emitter. *See* Light(s)
- Energy balance, 862
- Energy conservation
- BRDF, 350
- Environment camera, 375–377
- defined, 375
 - ray coordinate computation, 376
 - rendering example, 377
 - See also* Cameras
- Environment maps
- changing, 739
 - for illumination, 740
 - See also* Infinite area lights
- Environmental projection, 376
- Equation of transfer, 888–891
- defined, 888
 - incident radiance, 889
 - radiance effects, 889
- Error reporting, 1068
- Euler–Poincaré formula, 153
- Even vertices, 194
- boundary rule application for, 198
 - one-ring rule for, 195
- EWA. *See* Elliptically weighted average
- Exit pupil, 389–394
- Exitant radiance, 339–341
- See also* Radiance
- Expected values, 750, 751, 764
- Exponent (floating point), 208
- Exponential distribution, 756
- Extinction coefficient, 696–697, 896, 928
- Extraordinary vertices
- defined, 185
 - determination, 189
 - See also* Vertices
- F**
- False sharing, 1083
- fBm. *See* Fractional Brownian motion
- Film, 483–500
- effect of, 483
 - image output, 494–496
 - implementations, 483
- Film tile
- creation, 488
 - defined, 489
- merging, 493
- parallel rendering, 27
- Filter(s)
- box, 475–477, 631
 - extent in each direction, 474
 - functions, 474–483
 - Gaussian, 477–479
 - Mitchell, 479–491
 - pbrt implementations, 474, 475
 - pixel reconstruction, 474
 - reconstruction, 626
 - separable, 478, 626–627
 - sinc, 473, 481–483, 405
 - support, 475
 - triangle, 477
 - unified resampling, 635
 - weight table computation, 487
 - width definition, 474
 - See also* Reconstruction
- Filter radius, 474
- Filter support, 474
- Final gathering, 971
- First fundamental form, 139
- Fixed-function graphics hardware, 1059
- Float file reader, 1070
- Floating point
- arithmetic, 213
 - IEEE standard, 208
 - “infinity”, 210
 - “not a number” (NaN) value, 210
 - representation, 208
 - See also* Rounding error
- Fluence, 396
- Fluence rate, 921
- Fluorescence, 334, 708
- Flux. *See* Radiant flux
- Focal distance, 369, 374
- Focal length, 368
- Focal point, 368, 386
- Foreshortening
- defined, 361
 - perspective projection, 365
- Forward differencing, 588
- Forward error analysis, 216
- Fourier analysis, 406
- concept, 403–405
 - defined, 402
 - equation, 405
 - use of, 403
- Fourier basis BRDF, 552–560
- accuracy considerations, 559
 - interpolation scheme, 556–558
 - representation, 553
 - sampling, 817–830
- Fourier material, 583–584
- Fourier pairs, 405

- Fourier series
 efficient evaluation, 559
 sampling, 828–830
- Fourier transform, 403–405
 convolution, 406
 defined, 403
 inverse, 406
 one-dimensional function, 406
 product of two functions, 406
 shah function, 406
- Fractional Brownian motion, 655–658
 antialiasing, 659
 graphs, 659
 implementation, 656
 noise, 658
 noise octave computation, 658
- Fragments
 \pm = symbol, 3
 content expansion, 3
 defined, 2
 initialization, 2–3
 page numbers, 4
 series of, 3
- Frames
 defined, 57
 standard, 58
 transformation of, 82
- Frequency domain
 defined, 403
 sampled signal, 407
 transforming to spatial domain, 404
- Fresnel equations
 for conductors, 517, 518, 520
 defined, 516
 for dielectric media, 520
 moments, 694–695
- Fresnel incidence effects, 549–552
 diffuse term derivation, 550, 551
 illustrated, 550
 spectra, 550
- Fresnel interface, 521
- Fresnel moment, 694
- Fresnel reflectance, 516–523
 computation for dielectric, 518
 for conductors, 518, 522
 for dielectric media, 518, 522–523
 for parallel polarized light, 518
 for unpolarized light, 518
See also Reflectance
- Fresnel transmittance, 694, 927
- Functions
 band-limited, 406, 409, 604
 clamping, 1062
 conditional density, 774
 decomposition, 3
 expected value, 750
- fractional Brownian motion, 656
 frequency, 404
 frequency space representation, 404
 image contribution, 1026
 image texture, 618
 importance, 964
 marginal density, 774
 noise, 648–664
 phase, 678, 680–683
 piecewise-constant 1D, 757–760
 ray-casting, 862
 scalar contribution, 1028
 as series of fragments, 3
 smooth interpolating, 658
 turbulence, 658–660
 variance of, 750, 751
- G**
- Gamma correction, 621
- Gaussian filter, 477–479
 defined, 477
 graph, 478
 utility functions, 479
See also Filters; Reconstruction
- Gaussian lens equation, 369
- Gauss-Jordan elimination, 1081
- Generalized nth nearest-neighbor estimate, 969
- Generator matrix, 457
- Geometric attenuation, microfacet, 546
- Geometric classes, 57
- Geometric distribution, 8
- Geometric optics
 defined, 334
 linearity assumption, 334, 349
 as reflection model source, 508
- Geometric primitives, 250–251
 allocation, 250
 with animated transformation, 251
 collection, 22–23
 methods, 251
- Geometric term. *See Geometry term*
- Geometry
 caching, 1053
 differential, 114, 117
 shading, 166–167
 specular transmission, 530
 surface, approximating, 602
- Geometry term
 cancellation, 875
 defined, 866
 generalized, 891, 1011
 in subsurface scattering material, 932
- Gibbs phenomenon, 414
- Glass material, 584, 686
- Global illumination
 algorithms, 862
 bidirectional path tracing, 990–1022
 Metropolis light transport, 1022–1042
 path tracing, 870–879
- Global include files, 1061, 1068
- Glossy specular reflection
 defined, 508
 illustrated, 509
See also Specular reflection
- Goniophotometric diagram lights, 728–731
 defined, 728
 example, 729
 for real-world light sources, 729
 scene rendering, 729
- GPU ray tracing, 1056
- Graphics state, 1118–1119
 current, 1118
 defined, 1118
 hierarchical, 1118–1119
 methods, 1125
 pushed, 1129
 storage, 1118
- Gray code, 459
 first values, 460
- Grid density medium, 690–692
- Grosjean diffusion solution, 924
- H**
- Half-angle
 Jacobian, 812–813
 vector, 545, 809
- Halton sampler, 450–454
 stratified sampler comparison, 462
- Halton sequences, 443–450
 characteristics, 444
 discrepancy, 444
 drawbacks, 448
 points, 445
 sample generation with, 462
 used by SPPM, 984, 988
- Hammersley sequences, 443
 drawbacks, 448
 points, 445
- Handedness. *See* Coordinate systems
- Head model, subsurface material, 585
- Hemispherical-directional reflectance, 514, 533
 estimation, 831
- Hemispherical-hemispherical reflectance, 515
 estimation, 831
- Lambertian BRDF, 533

- Henyey–Greenstein phase function, 682
 after n scattering events, 918
 asymmetry parameter, 681, 682
 defined, 681
 plot, 682
 sampling, 899
 weighted sum, 683
- Heterogeneous scattering, 901
- Heterogeneous volumes, 672, 688–689, 893–894
- Hierarchical graphics state, 1118–1119
- Hierarchical linear bounding volume hierarchies. *See* Linear bounding volume hierarchies
- Histogramming, 967
- Homogeneous coordinates, 82–84
 application, 93–95
 composition, 96
 defined, 82
 look-at transformation, 91
 rotation, 88–91
 scaling, 87
 transformation, 82–84
 translation, 85
 weight, 82
- Homogeneous volumes, 672, 893–898
- Hyperboloids, 152
- I**
- Ideal sampling, 405–409
- Identity matrix, 84, 94
- Identity transformation, 84
- IEEE floating-point standard, 208
- Illumination algorithms, 862
- Image(s)
 filtering, 472
 reconstruction, 472–483
 resampling, 626
 resizing, 626
- Image contribution function, 1026
- Image file input/output, 1066–1068
- Image pixels, 402
- Image pyramids
 defined, 625
 EWA filter, 636
 implementation, 626
- Image sampling interface, 416–432
- Image textures, 618–639
 defined, 618
 example illustration, 618
 function value computation, 634
 isotropic triangle filter, 632–634
 memory requirements, 619
 memory use, 630
 MIP maps, 623–632
- parameterization, 619
See also Texture(s)
- Implicit form
 cones, 150
 cylinders, 143
 hyperboloids, 152
 paraboloids, 151–152
 spheres, 131
- Importance functions
 in particle tracing, 964
 of perspective camera, 953
- Importance sampling, 794–801, 880
 defined, 792
 direct lighting, 856
 ease of use, 797
 Monte Carlo estimator convergence, 794
 multiple, 797–799
 variance and, 787
- Incident radiance, 339–341, 343, 526, 854
 as continuous function, 413
 equation of transfer, 889
- Include files
 global, 1061, 1062
 main, 1061–1062
- Index of refraction
 for conductors, 517
 defined, 516
 of gold, 521
- Indirect light transport, 11
- Infinite area lights, 737–741, 959–960
 defined, 737
 for environment lighting, 737
 illumination illustration, 737
 PDF initialization, 738
 projection, 959–960
 ray PDF computation, 959
 sample point conversion to direction, 849
 sampled direction PDF computation, 849
 sampling, 845–851
 sampling steps, 845
 total power, 741
See also Area lights; Light sources; Light(s)
- Initialization
 APIs, 1110, 1112
 with CTM, 1116
 current graphics state, 1118
- In-scattering, 678–680
 defined, 676
 illustrated, 680
 phase function, 689
 radiance increase, 679
- source term, 690
See also Scattering
- Instancing, object, 251–254
 defined, 251
 memory use, 251
 shapes with, 247
- Integrals
 direct lighting, 856–861
 estimating with Metropolis sampling, 771
 over area, 347–348
 over paths, 866–868
 over points, 867
 over projected solid angles, 343–344
 over spherical coordinates, 344–347
 radiometric, 343–348
 XYZ color, 322
- Integrands
 delta distributions in, 868–869
 partitioning, 869–870
- Integrators
 defined, 805
 direct lighting, 851–861
 interfaces, 24, 25
 path, 866–867, 873
 path tracing, 900
 surface, 33, 805
 for Whitted ray tracing, 32–38
- Integro-differential equation, 888, 920
- Intensity, 337–339
 defined, 337
 equation, 339
 solid angle and, 337–339
- Interaction
 interface, 114–116
 medium, 687–688
 surface, 116–120
- Interior BVH nodes, 260, 282, 283
- Interior kd-tree nodes
 field usage, 286
 information access, 286
 initialization, 290
 memory, 286
 processing, 299
 storage, 287
- Interior vertices, 194
 limit rule, 203
 one-ring vertices for, 196
 tangent computation, 204
 updating positions of, 194
- Interpolation
 barycentric, 162
 keyframe matrix, 97, 98
 MIP map level, 632
 rotations, 98, 107
 scale, 98

Interpolation (*continued*)

- scale matrix, 107
 - SPDs, 317
 - spherical linear, 101–103
 - translation, 98
- Interreflection
- microfacet models and, 534
 - Oren-Nayar model and, 535

Intersections

- aggregate, 255
- error bounds, 222–229
- existence of, 24
- information, 130
- ray-cylinder, 144–146
- ray-disk, 148–149
- ray-object, 7–8
- ray-plane, 127, 602
- ray-primitive, 24
- ray-shape, 129
- ray-sphere, 134–138
- ray-triangle, 157–166
- routines, reading/writing, 130
- shape interface, 130

Interval arithmetic, 111

Inverse Fourier transform, 405

Inverse transformations, 82

Inversion method, 753–760

- defined, 755
 - exponential distribution example, 756
 - generalization of, 772
 - piecewise-constant 1D functions, 757–760
 - power distribution example, 755–756
 - use illustration, 754
- See also* Random variables

Irradiance, 336–337

- defined, 336
- differential, 9, 349
- integral, 344–347
- Lambert’s law, 337
- at points, 343
- vector, 921

Isotropic phase functions, 680

Isotropic reflection functions, 508–509

Isotropic triangle filter, 632–634

K

kd-tree accelerator, 284–302

kd-trees, 1027–1033

- as binary trees, 286
- building of, 289
- child node initialization, 290
- construction, 288–297
- defined, 284

depth, 288, 290, 292

- illustrated, 285
 - interior nodes, 286
 - intersection tests, 301–302
 - leaf nodes, 286
 - node allocation, 290
 - node array, 298
 - node, bounding boxes overlapping, 295
 - node children pointers, 300
 - node layout, 287
 - node memory, 287, 288
 - node storage, 286
 - node surface area, 294
 - node type decision, 290
 - primitive indices, 289
 - representation, 286–288
 - split axis selection, 293
 - split preference, 291
 - traversal, 297–302
 - tree-building algorithm, 291
 - working memory allocation, 289, 297
- See also* Acceleration structures

Kernel methods, 968

Keyframe matrices, 97, 98

Kirchoff’s law, 711

L

Lambertian reflection, 532–533

BRDF, 532

See also Reflection

Lambert’s law, 337

Lanczos filter. *See* Sinc filter

Lanczos window, 481

Laplace operator, 922

Latin hypercube sampling (LHS)

advantage, 439

defined, 438

illustrated, 439

sample generation, 440

sample permutation, 440

stratified sampling versus, 438

See also Sampling

Leaf BVH nodes, 258, 259, 283

Leaf kd-tree nodes

creation, 289

field usage, 286

initialization, 287

intersection tests, 301–302

memory, 286

overlapping primitives, 287

primitive checking in, 301

primitive id storage, 287

storage, 287

Left-handed coordinate system

- defined, 58
 - illustrated, 59
 - transformations changing, 96
- Lens element, 377
- Lens system, 379
- exit pupil, 389
 - f-number, 372
 - focal distance, 369
 - focal length, 368
 - focal points, 386
 - focusing, 388
 - optical axis, 379
 - principal planes, 386
 - spherical aberration, 386
 - tabular description, 380
 - thick lens approximation, 386–388
 - tracing rays through, 382

LHS. *See* Latin hypercube sampling (LHS)

Light(s)

area, 733–737, 959–960

attenuation, 13

behavior at wavelengths, 334

blackbody, 709

cosine falloff, 9

directional distribution, 339

distant, 731–733

energy, 8

fluorescence and, 334

goniophotometric, 728–731, 956

infinite area, 737–741

leaks, 574

linearity behavior, 334

multiple, 9

phosphorescence and, 334

physical processes, 708

point, 719–731, 836

polarization and, 334

projection, 724–728, 956

propagation, simulating, 25

sampling, 835–836, 845–856

with singularities, 836

spotlights, 15, 721–724, 956

steady state behavior, 334

texture projection, 724–728

total emitted power, 717

Light distribution, 8–10

after many scattering events,
919

defined, 4

geometric, 8

in media with high albedos, 917

radiometric, 8

spotlight, 15

- Light sampling, 835–963
 area lights, 957
 goniophotometric lights, 956
 infinite area lights, 959
 lights with singularities, 836
 methods, 835
 point lights, 719, 836, 955
 projection lights, 956
 spotlights, 721–724, 956
- Light sources, 707–745
 area lights, 8, 733–737, 957
 distant lights, 731–733
 goniophotometric lights, 728–731, 956
 implementations, 707
 incident direction, 716
 infinite area lights, 737–741, 959–960
 interface, 707
 intersection between ray and, 10
 isotropic, 719
 point lights, 339, 836
 primitives as, 250
 projection lights, 724–728, 956
 sample default number, 714
 sampling, 835–851
 scene, 23
 scene definition, 1123
 spotlights, 721–727
 surface energy, 10
 texture projection lights, 724–728
 total power emitted, 717
 visibility testing, 717–719
- Light subpath, 991
- Light transport equation (LTE), 861–870
 analytic solutions, 863–865
 basic derivation, 862–863
 conservation of power, 862
 defined, 12
 delta distribution in integrand, 868–869
 energy balance, 862
 equation of transfer and, 887
 evaluation of, 862
 integral over paths, 866–868
 measurement equation and, 948–963
 partitioning integrand, 869–870
 path integral form, 866–868
 path tracing, 870–879
 surface form, 865–866
 volume, 15
- Lighting
 direct, 36, 851–861
- Linear bounding volume hierarchies (LBVs)
 defined, 268
 flattening, 280–282
- hierarchical, 270
 treelets, 270
- Linear interpolation, 1062
- Linear transformations, 81
- Linearity
 geometric optics, 349
 light behavior, 334
- Literate programming, 1–4
 cross-referencing, 3–4
 defined, 1
 as enhanced macro substitution package, 3
 features, 2
 indexing, 3–4
 influences, 2
- Load balancing, 27–38
- Local exploration, 1022
- Local illumination algorithms, 862
- Local support, 195
- Lock-free algorithm, 1085
- Look-at transformation, 91–93
 defined, 91
 finding entries of, 92
See also Transformations
- Loop subdivision surfaces, 182
 local support, 195
 manifold control mesh, 185
 modified rules, 192
 refinement process, 184
 rules, 184
 scheme, 185
 vertices, 182
 weights, 182
- LTE. *See* Light transport equation
- Luminance, 341–343
 defined, 341
 representative values, 342
 spectral response curve and, 341
 units, 341
- Luminous efficacy, 708–709
- M**
- Machine epsilon, 214
- Macrosurface, 534
- Mailboxing, 255, 310
- Main function, 22
- Main rendering loop, 26–32
 class relationships, 26, 27
 defined, 26
- Malley's method, 779
- Marble, 663–664
 defined, 663
 illustrated, 663
 parameters, 663
- See also* Noise; Texture(s)
- Marginal density function, 774
- Markov chain, 1022
- Masking, microfacet models, 534, 541
- Materials, 571–594
 bump mapping, 584–590
 default, 1112
 description, 1121–1123
 displacement function association, 586
 glass, 584
 interface/implementations, 577–584
 matte, 578–579
 metal, 584, 585
 mirror, 584
 mix, 581–583
 parameters, 1119–1122
 plastic, 579–581
 shape, 1125
 substrate, 584
 subsurface, 584, 585
 translucent, 584
 uber, 584
- Matrix
 decomposition, 98
 orthogonal, 88
 row-major layout, 83
See also Transformations
- Matte material, 578–579
 defined, 578
 texture evaluation, 579
See also Materials
- Maximized minimal distance sampler
 defined, 465
 generator matrices of, 466
- Maxwell's equations, 334
- Mean free path, 676
- Measurement equation, 948–949
 camera, 395–397
 defined, 948
 in pixel measurements, 949
- Medium interaction, 687–688
- Medium interface
 defined, 684
 sampling, 891
- Memory
 atomic operations, 41–42, 1084–1085
 barrier instructions, 1082
 blocked layout, 1076
 BSDF, management, 576
 cache-friendly usage, 1071–1073
 coherence models, 1082–1084
 coherent shared, 39
 image maps, 619, 630
 kd-tree nodes, 286–289

- Memory** (*continued*)
 - linear layout of BVH in, 280
 - object instancing and, 251
 - overhead, reducing, 1053
- Memory allocation**
 - arena-based, 1073–1076
 - BSDF, 249
 - functions, 1072
 - variable stack, 1070–1071
- Memory arenas**, 1073–1076
 - defined, 1073
 - function, 1074
 - getting block of memory for, 1075
- Memory management**, 1070–1079
 - arena-based allocation, 1073–1076
 - blocked 2D arrays, 1076–1079
 - cache-friendly memory usage, 1071–1073
 - complexity, 1070
 - variable stack allocation, 1070–1071
- Metal material**, 584, 585
- Metamers**, 322
- Method of images**, 925
- Metropolis Light Transport (MLT)**, 1022–1042
 - advantages, 1022
 - bootstrapping, 1040
 - defined, 1022
 - large step mutation, 1024
 - light-carrying paths, 1022
 - local exploration, 1022
 - path tracing comparison, 1026, 1027
 - small step mutation, 1024
- Metropolis sampling**, 762–771
 - acceptance probability, 763
 - algorithm pseudo code, 764
 - basic algorithm, 763–764
 - defined, 762
 - detailed balance property, 763
 - disadvantages, 763
 - estimating integrals, 771
 - expected values, 764
 - graph illustration, 767
 - mutation strategies, 764–765, 767
 - one-dimensional setting example, 766–770
 - PDFs and, 765
 - sample generation, 768
 - start-up bias, 766
 - stationary distribution, 763
 - strategies comparison, 769
 - See also* Sampling
- Metropolis-Hastings algorithm**, 762
- Microfacet distribution**
 - Beckmann–Spizzichino, 538, 808–809
 - defined, 537
- Microfacet**
 - criterion, 537–538
 - Trowbridge–Reitz, 540
 - visible area, 808, 810–811
- Microfacet masking**, 541
- Microfacet models**, 533–549
 - components, 534
 - defined, 533
 - function, 533
 - geometric attenuation, 546
 - geometric effects, 534
 - interreflection and, 534
 - masking and, 534
 - Oren–Nayar, 534–537
 - reflection computation, 534
 - shadowing and, 534
 - Torrance–Sparrow, 544–549
 - See also* Reflection; Reflection models
- Microfacet shadowing**, 541
- Microfacets**
 - defined, 533
 - differential area of, 546
 - half-angle Jacobian, 812–813
 - half-angle vector, 545
 - surface comprised of, 533
- Microsurface**, 534
- MIP maps**, 623–632
 - box filter, 631
 - defined, 623–624
 - filter weights, 628
 - image pyramid, 625, 626
 - image resampling weights computation, 626
 - level computation, 632
 - level initialization, 630
 - level selection, 632
 - level trilinear interpolation, 633
 - projection light, 726
 - reconstruction filter, 626
 - texels, 627, 630
 - See also* Image textures
- Mirror material**, 584
- Mitchell filter**, 479–481
 - defined, 479
 - graph, 478
 - illustrated, 480
 - negative lobes, 479
 - one-dimensional filter function, 479
 - ringing/blurring trade-off, 479
 - See also* Filters; Reconstruction
- Mix material**, 581–583
 - defined, 581
 - See also* Materials
- Mix textures**, 616–617
 - defined, 616
 - evaluation, 617
 - See also* Texture(s)
- MLT**. *See* Metropolis Light Transport
- Möbius strip**, 187
- Modulus function**, 1062
- Moment**, 921
- Monopole**, 923
- Monte Carlo estimator**, 751–753
 - convergence, 753
 - convergence rate, 753
 - defined, 751
 - efficiency, 787
 - error, 753
 - evaluation, 753
 - expected value, 751
 - extending to multiple dimensions, 752
 - multiple import sampling, 797
 - for scattering equation, 815
- Monte Carlo integration**, 747–885
 - background, 748–751
 - concepts, 747–801
 - defined, 748
 - disadvantage, 748
 - efficiency improvement, 805–885
 - randomness, 747
- Morton code**, 268
- Motion blur**, 356
- Multidimensional transformations**, 773–787
 - conditional density function, 774
 - cosine-weighted hemisphere sampling example, 779–780
 - hemisphere sampling example, 774–775
 - piecewise-constant 2D distributions example, 784–787
 - sphere sampling example, 775–776
 - triangle sampling example, 781–782
 - unit disk sampling example, 776–779
 - See also* Transformations
- Multiple importance sampling (MIS)**, 797–799
 - balance heuristic, 798
 - in BDPT, 1012
 - defined, 798
 - direct lighting, 858
 - illustrated, 857, 1012
 - Monte Carlo estimator, 798
 - power heuristic, 799
 - sample weighting, 798
 - variance reduction, 798
 - See also* Importance sampling
- Multiple scattering**
 - effect of, 901
 - monopole solution, 923
 - relation to single scattering, 936
- Multiplexed MLT (MMLT)**, 1025
- Multipoles**, 941

Mutation strategies, 764–765, 767
perturbation, 765, 1022, 1024

Mutexes, 40–41
 reader-writer, 1085

N

Nearest-neighbor techniques, 968
Newton-bisection, 825
 illustrated, 827
Noise, 648–664
 as band-limited function, 655
 for fractional Brownian motion, 656
 frequency content, 655
 functions, 648–649
 idioms, 655–660
 marble, 663–664
 octave of, 656
Perlin, 649–653
 random polka dots, 653–655
turbulence, 658–660
value, 649
windy waves, 662–663

Non-symmetric scattering, 960–963
 due to refraction, 961
 due to shading normals, 962

Nonuniform sampling, 411, 472

Nonuniform scaling, 87

Normalization
 CDF, 758
 vector, 65–66

Normalized device coordinate (NDC)
 space, 359, 377

Normalized floating-point number, 208

Normals, 71–72

 for closed shapes, 118
 defined, 71
dot product computation between, 72
implementation, 71
microfacet, 533
orientation, reversing, 118
partial derivative of, 138–140
shading, 166, 574
transforming, 93–95

Noweb system, 2

n-rooks sampling. *See* Latin hypercube sampling (LHS)

Nyquist frequency, 412, 472

Nyquist limit, 412, 623, 658

O

Object(s)

 foreshortening, 361, 365
 implementations, adding, 1131
 instances, 1127

layout in memory, 1071
scene, 1130
Object instancing, 251–254
 defined, 251
 memory use, 251
 in scene description, 1128–1129
 shapes with, 247

Object space
 defined, 124, 358
 spheres in, 132
 See also Coordinate spaces

Object subdivision, 254

Octave, noise, 656

Octrees
 defined, 284
Odd vertices, 194
Offset rays, 600
One-ring
 for boundary vertex, 197
 defined, 196
 for even vertex, 196
 for interior vertex, 196
 rule, 195

One-to-one transformations, 82

Open mesh, 185

OpenEXR, 1067

OpenGL, 1103

Optical axis, 379

Optical radius, 697

Optical thickness, 678

Optimizations
 aligned memory, 1071–1072

 blocked arrays, 1076–1079

 division by integer constants, 447

 fast math, 211

 mailboxing, 255, 310

 memory arena, 1073–1076

 overview, 46–47

 parallelism, 1082–1093

 retrospective, 1051–1059

Oren–Nayar diffuse reflection, 534–537

 cosine term computation, 536

 defined, 534

 shadowing, masking, interreflection
 and, 535

 sine and tangent term computation,
 536

See also Microfacet models; reflection

Organization, this book, 44–45

Oriented bounding boxes (OBBs), 76

Orthogonal matrix, 88

Orthographic camera, 361–364

 church model image, 362

 defined, 361

 depth of field, 363

 differential rays computation, 363

illustrated, 363
ray creation, 363
viewing transformation, 362
See also Cameras; Projective camera
models

Orthographic projection, 363

Out-of-gamut colors, 495

Out-scattering, 676–678
 beam transmittance, 677–678
 defined, 676
 illustrated, 677
 probability, 676
 radiance and, 676
 See also Scattering

P

Packet tracing, 1057

Paraboloids, 151–152

Parallelism, 1082–1093

 atomic operations, 1084–1085
 for loops, 1087–1093
 memory coherence models and
 performance, 1080–1082
 mutexes, 40–41, 1084
 task system, 1101

Parallelization, 38–44

 challenge, 38–39

 conventions, 42–43

 data races and coordination, 39–42

See also pbrt rendering system

Parallelograms, 65

Parameter sets, 1105–1109

 adding to, 1105–1106

 defined, 1105

 item structure, 1106

 methods, 1108

 parameters, 1105

 values, looking up, 1107–1109

Parametric form

 cones, 150–151

 cylinders, 142–143

 disks, 147

 hyperboloids, 152

 paraboloids, 151–152

 rays, 7, 72

 spheres, 131

 triangles, 163–164

Partial cylinders, 142–143

Partial derivatives

 cone, 150–151

 cylinder, 142–143

 disk, 147

 estimation with offset rays, 600

 hyperboloid, 152

 paraboloid, 151–152

Partial derivatives (*continued*)

- shading normal, 119
- sphere, 137–138
- triangle, 163–164

Partial spheres, 136–137

Participating media, 671

Particle tracing, 949, 963–966

- defined, 963
- illustrated, 963
- samples of illumination, 963
- theoretical basis, 963–966

Particles

- distribution, 974
- intersections, 985
- weights, 964, 974

Partitioning integrands, 869–870

Partitioning primitives, 256, 260, 264

- with approximate SAH, 265
- axis selection, 258
- bucket boundary, 267
- goal, 261
- interior nodes, 260
- through node midpoint, 261–262

See also Bounding volume hierarchies (BVHs)

Path(s)

- camera, 991
- connecting, 991
- contributions, 866–868
- incremental construction, 874–875
- integral over, 866–868
- light, 991
- mean free, 676, 701, 906, 928, 938
- particle, 708
- photon, 983–990
- ray, vertex generation, 997
- sampling, 873–874
- termination, 788, 878, 989
- throughput, 868
- throughput weight, 876

Path space

- defined, 866–868
- generalized, 890–891
- in Metropolis Light Transport, 1022
- versus primary sample space, 1044

Path tracing, 870–879

- bidirectional, 990–1021
- defined, 870
- implementation, 875–879
- incremental path construction, 874–875
- integrator, 870
- Metropolis Light Transport (MLT) comparison, 1026–1027
- overview, 872–873
- path sampling, 873–874

scene rendering, 870

- pbrt rendering system
- abstract base classes, 15
- bugs, 64
- code use, 48–53
- conventions in, 42–43
- interface types, 16
- main() function, 16, 19–23
- main rendering loop, 26–32
- overview, 15–38
- parallelization of, 38–44
- parsing phase, 17
- phases of execution, 17–19
- rendering loop, 18
- scene file format, 17
- scene representation, 19–24
- source code distribution, 16

PCG random number generator, 1065

PDF. *See* Probability density functions (PDFs)

- Perfect specular reflection
- defined, 516
- illustrated, 517

See also Specular reflection

Perlin noise, 649–653

- characteristics, 649
- computation, 649
- data, 652
- defined, 649
- gradient weights computation, 649, 651
- illustrated, 650
- implementation, 649, 652
- nested permutations, 652
- weights interpolation computation, 651

See also Noise

Perspective camera, 365–368

- implementation, 365
- rendering example, 365

See also Cameras; Projective camera models

Perspective projection

- distances/angles and, 365
- foreshortening, 365
- offset ray computation, 368
- parallel lines and, 365
- perspective viewing, 365
- ray origination, 364
- transformation matrix, 366
- transformation steps, 366–367

Perturbation, 765, 1022, 1024

- Phase functions, 678, 680–683
- anisotropic, 680
- defined, 680
- direction convention, 681

Henyey–Greenstein, 681–683

- implementations, 683
- isotropic, 680
- probability density for sampling, 898–899
- reciprocal property, 680
- sampling, 873–874
- weighted sum, 683

Phenomenological reflection models, 507

Phosphorescence, 334

Photometry, 341

Photon beam diffusion, 916

- multiple scattering term, 916–930
- single scattering term, 930–933

Photon mapping

- bias, 966
- defined, 966–972
- density estimation, 967–970
- direct lighting, 970
- final gathering, 971, 1042–1043
- indirect lighting, 970
- kernel, 968

Photons

- close to lookup point, 969
- distribution, 963
- for final gather ray, 1043
- interpolation, 967–969
- scattering, 987
- weights, 963, 987

Photorealistic rendering

- goal, 4
- history, 48–52
- ray-tracing algorithm and, 4–15

See also Rendering

Physical (wave) optics, 508

Piecewise-constant 1D functions, 757–760

Piecewise-constant 2D functions, 784–787

- illustrated, 786

Pincushion distortion, 355

Pinhole cameras

- defined, 5
- depth of field and, 368–375
- elements, 5
- illustrated, 6
- simulation, 6

See also Cameras

Pixels

- addressing, 415
- coordinates, 491, 628
- discrete integer, 415
- display, 402
- display values and, 621
- image, 401

- measurements, 948–949
 reconstruction filter, 474
 understanding, 415–416
- Planar mapping, 613–614
- Plane, 127
 intersection test, 127
- Planck's law, 709
- Plastic material, 579–581
 Dragon rendered with, 580
 modeling, 579
 parameters, 579
See also Materials
- Point lights, 719–731
 goniophotometric diagram, 728–731
 intensity, 339
 positioning, 719
 projection, 719
 sampling, 836
 spotlights, 721–724
 texture projection, 724–728
 total power emitted, 721
See also Light sources; Light(s)
- Points, 67–71
 adding, 69
 auxiliary intersection, 602
 barycentric coordinate computation for, 162
 bounding box, 78
 in camera space, 363, 366
 defined, 67
 denoting, 67
 distance between, 70
 Halton sequence, 443–444
 Hammersley sequence, 445
 homogeneous, 82–84
 irradiance at, 343
 optical thickness between, 678
 samples, 403, 411
 sphere sample, on spherical light source, 842
 subtracting, 69
 transforming, 93–97
 translation and, 85
 translucent surfaces, 692
 weighted sums of, 70
- Poisson disk patterns, 421
- Polar coordinates, 772–773
- Polar decomposition, 104
- Polarization, 334
- Polka dots, 653–655
 antialiasing, 654
 application illustration, 654
 center, 655
 declaration, 617
 indices computation, 654
 radius, 655
- Postaliasing, 410
- Power distribution, 755–756
- Power heuristic, 799, 1012
- Prealiasing, 410
- Prefiltering, 412–413
- Primary sample space MLT (PSSMLT), 1023
- Primary sample space, 1024
 sampler, 1028–1035
- Primitive interface, 248–254
 defined, 250
 methods, 250–252
- Primitive subdivision, 255
- Primitives
 animated, 251–254
 as area light sources, 250
 centroids, computing bounds of, 259
 geometric, 250–251
 groups, rejection of, 254
 instanced, 252
 overlapping, kd-tree nodes, 286
 splitting based on midpoint centroids, 261
 surface scattering inside, 249
 surface shader bound to, 571
 transformation, 247
 world space bounds, 249
- Principal plane, 386
- Principle of similarity, 917–918
- Probability density. *See* Probability density functions (PDFs)
- Probability density functions (PDFs), 749–750
 CDF and, 749
 discrete, 754
 infinite area light, 959–960
 Metropolis sampling and, 765
 piecewise-constant 1D functions, 757–760
 power distribution, 755
 sampling direction towards spheres, 840–845
 sampling for infinite area light, 959–960
 shape sampling, 836–838
 spotlight illumination distribution, 836
- Procedural texturing, 640–648
 antialiasing, 640, 647
 checkerboard, 642–646
 defined, 640
 fractional Brownian motion for, 656
 implications, 640
 implicit pattern definition, 648
 UV texture, 641–642
See also Texture(s)
- Progressive photon mapping, 970–971
- Projected solid angles, 344–345
 determination, 343
 illustrated, 343
 integrals over, 343–344
See also Solid angles
- Projection
 infinite area lights, 959–960
 light sampling, 956
 light sources, 940–941
 matrix, 386
- Projective camera models, 358–375
 depth of field, 368–375
 implementation, 360
 orthographic, 361–364
 perspective, 365–368
 screen-to-raster, 360
 transformation matrix, 358
- Pseudo-random number generator, 1065–1066
- PSSMLT. *See* Primary sample space MLT
- Q**
- Quadratic equations, 135, 136, 1079–1080
- Quadrics
 checkerboard texture applied to, 642
 cone, 150–152
 cylinder, 142–146
 defined, 131
 disk, 146–150
 hyperboloid, 152
 paraboloid, 151–152
 polka dots applied to, 653
 sphere, 131–142
 surface area, 141–142
 UV texture applied to, 642
- Quasi Monte Carlo, 792
- Quaternions, 99–101
 addition, 100
 along animation path, 102
 defined, 99
 inner product, 100
 interpolation, 101–103
 multiplication, 99
 representation, 99
 rotation matrix, 100
 subtraction, 100
- R**
- Radiance, 339
 absorption and, 674
 defined, 339
 differential, 349

Radiance (*continued*)
 equation, 339
 exitant, 339–341, 865
 illustrated, 340
 impossible values, 31
 incident, 339–341, 343, 349, 854
 in-scattering and, 678
 outgoing, distribution, 711
 out-scattering and, 676
 over set of directions, 343
 properties, 339
 total, 36
 Whitted integrator evaluation, 33

Radiant exitance, 336

Radiant flux, 335
 defined, 335
 differential, 337
 illustrated, 336

Radiative equilibrium, 939

Radiative transfer, 334

Radical inverse
 computation, 444
 defined, 443
 inverse, 447
 permuted digits, 448
 positive integers, 443
 scrambled, 448
 sequences using, 448

Radiometric distribution, 8

Radiometric integrals, 343–348
 evaluation of, 343
 irradiance, 346, 347
 over area, 347–348
 over projected solid angle, 343–344
 over spherical coordinates, 344–347
 working with, 343–348

Radiometry, 334–343
 basic, 334–343
 defined, 313, 334
 energy, measurement of, 335
 flux, 335, 336
 geometric optics, 334
 intensity, 337–339
 irradiance, 336–337
 luminance, 341–342
 Maxwell’s equations and, 334
 power, measurement of, 335
 radiance, 339
 radiant exitance, 336
 radiative transfer, 334

Radiosity (rendering algorithm), 48

Radix sort, 274

Random number generator, 1065–1066

Random polka dots, 653–655

Random sampling, 791

Random variables
 applying functions to, 748
 BSDF, 832
 canonical uniform, 749
 CDF, 749
 continuous, 749–750
 defined, 748
 discrete, 749
 independent, 751
 inversion method, 753–760
 PDF, 750
 rejection method, 760–762
 sampling, 753–762
 transforming distributions, 771–773

Random walk, 1005

Randomness, 747

Raster space
 computing, 363
 sample point, 364
See also Coordinate spaces

Ratio tracking, 898

Ray differentials, 74–81
 in antialiasing textures, 38
 computation for specular reflection, 607
 defined, 30
 in finding filter regions, 605
 offset rays in, 606
 reflected, 606
 tracking, 606
 transmitted, 606

Ray marching, 894–895

Ray propagation, 13–15
 defined, 5
 participating media, 13

Ray tracing
 for BSSRDF sampling, 906–913
 defined, 4
 early example, 12
 environment camera, 375–377
 on GPUs, 1056
 high-performance CPU, 1057
 image projections, 375
 photorealistic rendering and, 4–15
 recursive, 5, 11–13
 single low-level shape representation, 1052
 strength of, 1053

Ray-casting function, 862

Ray-cylinder intersections, 144–146

Ray-disk intersections, 148–160

Ray-object intersections, 7–8
 acceleration structure, 8
 brute-force approach, 7–8
 defined, 4
 execution time, 254

point, 7

sphere, 140

Ray-plane intersections, 127, 602

Ray-primitive intersections
 information about, 249
 light-scattering properties, 249

Rays, 72–75
 camera simulation and, 6–7
 constructing, 74
 defined, 72
 denoting, 72
 “epsilon” value, 206–207
 illustrated, 73
 image location conversion to, 7
 importance, 949
 marching process, 895
 offset, 600
 parametric distance to intersection, 249
 parametric form, 7, 73
 reordering for coherence, 1053–1054
 shadow, 9, 24
 testing for intersection, 247
 time value, 73
 traversal through kd-tree, 298
 tree of, 13

Ray-triangle intersections, 157–166
 barycentric coordinates, 162
 cost of, 156
 distance along ray, 162

Read for ownership (RFO), 1083

Reader-writer mutexes, 1084

Realistic cameras
 defined, 377
 exit pupil, 389
 focusing, 388
See also Lens system
 ray generation, 394
 representation of, 379

Reconstruction, 472–483
 approximation, 402
 artifacts, 410
 box filter, 475–477
 defined, 402
 filter functions, 474–483
 Gaussian filter, 477–479
 ideal, 405–409, 472
 of images, 472
 Mitchell filter, 479–481
 pixel filter, 473
 pixel values, 473
 postaliasing, 410
 sinc filter, 481–483
 triangle filter, 407, 477

Recursive ray tracing
 defined, 5

- illustrated, 13
- mirror-reflection direction, 12
- tree of rays, 13
 - See also* Ray tracing
- Reduced albedo, 918
- Reduced attenuation coefficient, 918
- Reduced extinction coefficient, 918
- Reduced scattering coefficient, 918
- Reflectance, 514–515
 - conversion process for, 330–331
 - distribution function, 508
 - estimation, 830–832
 - Fresnel, 516–522
 - hemispherical-directional, 514
 - hemispherical-hemispherical, 515, 831
 - Lambertian surface, 864
 - properties, 507
- Reflected radiance, 350
 - See also* Radiance
- Reflection
 - anisotropic, 508
 - BRDF, 349–351
 - BSSRDF, 351
 - categories, 508
 - computations, 509
 - coordinate system, 509
 - diffuse, 508, 509, 534–537
 - direction, 12
 - distribution properties, 507
 - equation, 350
 - glossy specular, 508, 509
 - isotropic, 508
 - Lambertian, 532–533
 - mechanisms, 349
 - Oren–Nayar, 534–537
 - perfect specular, 508, 509
 - retro-reflective, 508, 509
 - specular, 36, 517, 523–526, 815–817
 - subsurface, 692
 - surface, 348–351, 805–885
 - total internal, 520
- Reflection models, 507–568
 - basic interface, 512–515
 - geometric setting, 509–512
 - implementations, 508
 - microfacet, 533–552
 - sources, 507–508
 - terminology, 508–509
- Refraction
 - index of, 516
 - indices for objects, 518
 - specular, 517
- Regular tracking, 894
- Regular vertices, 182
- Rejection method, 760–762
 - defined, 760
- efficiency, 760
- example, 761
- sample generation, 761
- sampling, 760
 - See also* Random variables
- Relative error, 215
- Rendering
 - aliasing sources, 414–415
 - APIs, 1103–1104
 - cleanup after, 1130
 - defined, 1
 - equation. *See* Light transport equation loop, 17
 - main loop, 26–32
 - Metropolis Light Transport (MLT), 1022–1042
 - options, 1114–1115
 - photorealistic, 4–15
 - sample pattern effect on, 433
 - world end and, 1129–1131
- Reporting, 1068–1069
 - error, 1068
 - progress, 1069–1070
- Retro-reflective reflection
 - defined, 508
 - illustrated, 509
- Reyes, 50
- RGB color, 325–331
 - coefficients, 328
 - conversion, 332
 - conversion table computation, 329
 - conversion to SPDs, 329
 - conversion from XYZ values, 327
 - conversion to XYZ values, 328
 - device dependent, 495
 - emission curves, 326
 - image film, 485
 - LED/LCD displays, 325
 - output values, 495
 - reflectance spectrum conversion, 330
 - SPD display, 327
 - spectrum creation, 332
 - spectrum functions computation, 324
 - values, 324, 328
- See also* Spectral representation
- Right-handed coordinate system, 58, 59
- Ringing
 - defined, 414, 479
- Rotations, 88–89
 - angle, 89, 90
 - arbitrary axis, 89–91
 - defined, 88
 - extracting, 105
 - interpolation, 98, 106
 - orthogonal matrix, 88
 - properties, 88
- x axis, 88–89
- y axis, 89
- z axis, 89
- Rounding error
 - absolute error, 215
 - of arithmetic operations, 213
 - backward analysis, 216
 - catastrophic cancellation, 218
 - forward analysis, 216
 - managing, 206–235
 - propagation, 214
 - “ray epsilon”, 203, 206–207
 - relative error, 215
 - running error analysis, 218
- Running error analysis, 218–220
- Russian roulette, 787–789
 - applying, 788
 - defined, 787
 - in path sampling termination, 876, 879
 - variance and, 788
 - weights, 788

S

- SAH. *See* Surface area heuristic
- Sample(s)
 - (0,2)-sequence, 455
 - accessing, 423
 - adding to film, 490
 - bootstrapping, 1038
 - implementations, 416
 - LHS, 438
 - light source, 714
 - linear segments between, 320
 - nonuniform distribution, 411–412
 - point, 403
 - position, 402
 - spacing, 599
 - SPD, 318, 319
 - spectral data, 319
 - value, 402
 - warping, 792–793
- Sample placement, 789–793
 - quasi Monte Carlo, 792
 - stratified sampling, 789–792
 - warping samples and distortion, 792–793
- Sample points
 - raster space, 363
- Sampled spectrum
 - arithmetic operations, 319
 - XYZ matching functions computation, 323
- Sampler(s)
 - (0,2)-sequence, 454–465

- Sampler(s) (*continued*)

adaptive, 412

basic interface, 421–425

global, 428

Halton, 441–454

Maximal minimum distance, 465–467

Sobol', 467–472

Stratified, 434–441
- Sampling

adaptive, 412, 497–498

aliasing and, 402, 409–410

antialiasing and, 410–412

approximation, 402

area lights, 442, 836–845

artifacts, 410

BSDFs, 832–835

BxDFs, 806–807

cosine-weighted hemisphere, 779–780

cylinders, 839

disks, 776–779, 838

frequency minimum, 410

FresnelBlend class, 814

goniophotometric lights, 956

ideal, 405–409

image interface, 416–432

image synthesis application, 413–414

importance, 794–799, 880

infinite area lights, 845–851

Latin hypercube, 439

low-discrepancy, 441–472, 498–499

method comparison, 436

Metropolis, 762–771

nonuniform, 411–412, 472

paths, 873–874

pattern comparison, 434, 435, 441

phase functions, 898–899

point, 411

point lights, 836

prealiasing, 410

process, 405, 406

random, 791

random variables, 753–762

reflection functions, 806–835

rejection, 760–762

shapes, 836–838

spheres, 840–845

spotlights, 956–957

stratified, 434–441, 789–792

texture, 599–604

theorem, 410

theory, 402–416

triangles, 781–782, 839–840

uniform, hemisphere, 774–776

volume scattering, 891–899
- Sampling rate

defined, 405
- low, 410

texture, 599–604, 623
- Saturation, 14
- Scalar contribution function, 1028
- Scale texture, 615–616

antialiasing and, 616

defined, 615

See also Texture(s)
- Scaling

BxDF adapter, 515

nonuniform, 87

properties, 87

transformation, 87–88

uniform, 87

vectors, 62
- Scattering

back, 682

defined, 673

equation, 350

in, 678–680

out, 676–678

subsurface, 351, 692–702, 903–939

surface, 5, 10–11

translucent media, 904

volume, 15, 671–704
- Scattering coefficient, 676

reduced, 918
- Scene definition, 1117–1131

hierarchical graphics state, 1118–1119

light sources, 1123

object instancing, 1127–1129

shapes and volume regions, 1123–1127

surface and material description, 1121–1123

texture and material parameters, 1119–1121

world end and rendering, 1129–1130
- Scene description

approaches, 1103–1104

defined, 16

initialization and rendering options, 1109–1117

interface, 1103–1104

new object implementations, 1131

parameter sets, 1105–1109

parsing, 16, 21, 30

scene definition, 1117–1131

state tracking, 1110–1111

transformations, 1111–1114
- Scenes

complexity, increased, 1053–1054

geometric object representation, 22–23

instance usage, 1127

light source, 23

representation, 18–24
- Screen space

decomposition, 26–28

defined, 358

See also Coordinate spaces
- Screen-to-raster projection, 360
- Second fundamental form, 139
- Self-illumination, 939
- Self-intersections, 207, 235
- Self-shadowing

microfacet models and, 534, 541–544

Oren–Nayar model and, 534–537
- Semi-infinite media, 917
- Separable BSSRDF, 693–696
- Separable filter, 478, 626–627
- Shading aliasing, 415
- Shading coordinate system, 509
- Shading normals

artifacts, 574

bump mapping and, 586

computation, 590

error types resulting from, 574
- Shadow rays

defined, 9

existence of intersections, 24

intersections, 255
- Shadows, 9
- Shah function

convolution, 406, 407

Fourier transform, 406
- Shapes, 123–244

abstraction of, 123

animated, 1123–1125, 1127

with animated transformation matrices, 247

API creation, 1131

bounding, 124–125

cone, 150–151

creation, 1066–1067

cylinder, 142–146

definition of, 124

disk, 146–150

hyperboloid, 152

interface, 123–131

normals, 118

with object instancing, 247

paraboloid, 151–152

parametric description, 116

sampling, 836–845

sidedness, 131

sphere, 131–142

surface area, 141–142

tetrahedron, 181

triangle, 152–167
- Shared data, updating, 42
- Sidedness, 131
- Significand (floating point), 208

- Simulation, 507–508
 Sinc filter, 409, 481–483
 finite extent, 483
 graphs, 482
 infinite support, 474
 Lanczos window, 483
 for resampling textures, 629
 ringing, 414, 474
 See also Filters; Reconstruction
 Single instruction, multiple data (SIMD)
 processing, 1055
 Single scattering albedo, 698
 relation to effective albedo, 936
 solving for, 938
 Single scattering term, 930–933
 Smith masking-shadowing function, 541
 Snell’s law, 516
 derivation of, 567
 for transmitted directions, 36
 Sobol’ sampler
 checkerboard structure, 469
 defined, 467
 generator matrices of, 468
 Sobol’ sequence, 455
 Solid angles
 defined, 337
 differential, 345–346
 illustrated, 345
 intensity and, 337–339
 measure relationship, 343
 projected, 343–344
 Solid checkerboard, 646–648
 defined, 646
 Solid textures, 640–648
 checkerboard, 646–648
 defined, 640
 representation problem, 640–641
 See also Texture(s)
 Spatial domain, 403
 convolution in, 406
 equivalent process, 409
 frequency space representation, 405
 functions, 405
 Spatial subdivision, 254
 Spectral matching curves, 322
 Spectral power distributions (SPDs)
 arbitrary, 322, 333
 basis function representation, 313
 coefficient computation, 315
 coefficient representation, 315
 computations with, 313
 debugging routine, 317
 defined, 313
 of fluorescent light, 314
 linear interpolation, 317
 real-world objects, 313
 representations, 323–324
 resampling, 320
 RGB color, 327
 samples, 318, 319
 Spectral representation, 313–318
 basis functions, 315
 class, 315–318
 design advantages, 315
 hiding details of, 315
 RGB color, 325–331
 Spectrum type, 315
 value zero, 316
 XYZ color, 322–325
 Spectral synthesis, 655–660
 Spectrum
 clamping, 317
 equality/inequality tests, 316
 initialization, 316
 luminance, 325
 sampled, 318–330
 shah function, 407
 square root, 317
 Specular reflection, 523–525
 BRDF, 523–524
 effect computation, 36
 direction computation, 525–526
 Fresnel modulation, 531–532
 implementation, 523
 ray differential computation for, 607
 sampling, 815–817
 texture filtering, 605, 606
 vector, 523, 524
 See also Reflection
 Specular refraction. *See Specular transmission*
 Specular transmission, 526–531
 BTDF for, 526, 527
 direction computation, 530–531
 effect computation, 36
 Fresnel modulation, 531–532
 geometry, 530
 modulation, 528
 sampling, 815–817
 vector, 528
 Spheres, 131–142
 basic setting, 132
 bounding, 133–134
 bounding box, 81
 center, 133, 840
 construction, 133
 defined, 131
 illustrated, 133
 implementation, 133
 implicit form, 132
 intersection, 134–138
 object space, 133
 parameterization, 132–133
 parametric definition, 137
 parametric form, 132, 135
 partial, 136–137
 partial derivatives of normal vectors, 138–140
 profile curve, 141
 radius, 133
 sample points, 842
 sampling, 840–845
 second fundamental form, 139
 spinning, 97
 surface area, 141–142
 texture mapping, 132, 133
 See also Shapes
 Spherical aberration, 386
 Spherical angles. *See Spherical coordinates*
 Spherical coordinates, 344, 773
 direction vectors and, 344
 formula, 377
 integrals over, 344–347
 Spherical linear interpolation (SLERP), 101–103
 Spherical mapping, 611–612
 Spherical moment, 921
 Spherical moments, 920
 Splatting, 494
 Spline
 defined, 167, 562
 interpolation, 171–172, 560–563
 sampling, 821–828
 weight computation, 562
 Splits
 access position, 292
 axis selection, 292
 edge, 199
 preferences, 291
 Splitting, 788–789
 data structures, 1072
 defined, 788
 integral estimator and, 789
 node plane, 299, 300
 primitives, 263
 surface area heuristic (SAH), 266–267
 Spotlights, 721–724
 angles, 722, 723
 defined, 721
 falloff computation, 724
 lighting distribution, 15
 sampling, 956–957
 scene rendering, 722
 strength computation, 723
 See also Light(s); Point lights
 SPPM. *See Stochastic progressive photon mapping*

- sRGB
 gamma curve, 621
 history, 665
 inverse curve, 622
 standard, 621–622
 from XYZ, 495
- Star discrepancy
 bounds, 420
 defined, 419
 of sequence, 420
- Start-up bias, 766
 defined, 766
 removing, 766
- State tracking, 1110–1111
- Stationary distribution, 763
- Steady state, light, 334
- Stefan–Boltzmann law, 710
- Stochastic progressive photon mapping, 971–990
 reflected radiance computation, 971
 visible point, 971, 976, 979–983
 update rules, 971–972
- Strata
 compact, 791
 defined, 432
 variance, 791
- Stratified sampling, 432–441, 789–792
 area light samples, 442
 curse of dimensionality, 433, 791–792
 example effect, 791
 function definitions, 437
 Halton sampler comparison, 451
- Latin hypercube sampling (LHS)
 versus, 438
- strata, 432
- uniform, over hemisphere, 795
- uniform random distribution versus,
 791
- utility routines, 437
- worst-case situation, 439
- See also* Sampling
- Subdivision, 181–206
 applying fixed number of times, 192
 on boundary edge, 198
 illustrated, 193
 levels of, 193
 limit surface and output, 203–206
 main loop, 193
 mesh topology update, 200–203
 object, 254
 primitive, 255
 refinement process, 184
 rule for edge split, 199
 rules, 181, 203
 rules application, 192
 spatial, 254
- steps, 181
 tetrahedron, 181
- Subdivision mesh
 boundaries, 185
 closed, 185
 consistently ordered, 186
 control, 181–183, 186
 extraordinary vertices, 185
 faces, 182, 184, 187
 in Loop subdivision scheme, 182
 manifold, 185, 186
 open, 185
 representation, 184–192
 split edges, 198
 topology update, 200–203
 triangular face pointers, 185
- Subdivision surfaces, 181–206
 advantages, 182
 applied to Killeroo model, 181, 183
 convex hull property, 195
 definition of, 181
 illustrated, 183
 Loop, 182
 mesh representation, 184–192
 methods, 182
 smooth, 182
 subdivision, 192–206
 triangle meshes, 155
See also Surfaces, Loop subdivision surfaces
- Substrate material, 584
- Subsurface scattering
 BSSRDF, 692–702
 illustrated, 904, 585
 materials, 584, 700–702
 sampling, 903–916
- Surface area
 cylinders, 146
 disks, 150
 kd-tree node, 289
 shapes, 130
 spheres, 141–142
 triangles, 160
- Surface area heuristic (SAH), 263–268
 computational expense, 264
 concept, 264
 defined, 263
 partitioning primitives with, 265
 splitting at bucket boundaries, 266–267
 splitting plane selection, 266.
See also Bounding volume hierarchies (BVHs); kd-tree accelerator
- Surface integrators
 class relationships, 33
See also Integrators
- Surface interaction, 116–120
- Surface normals. *See* Normals
- Surface reflection, 348–351
 BRDF, 349–351
 BSSRDF, 351
 modeling effects, 348
 translucent, 348
- Surface scattering, 10–11
 BRDF, 10
 BSDF, 11
 defined, 5
 determination, 10
 geometry, 11
 inside primitive, 249
- Surface shader, 508, 571
- Surfaces
 anisotropic, 509
 area density of flux arriving/leaving, 336
 composition, 507
 description, 1121–1123
 diffuse, 508, 509
 glossy specular, 508, 509
 implicit form, 131
 isotropic, 508–509
 light transport beneath, 351
 limit, 181
 microfacets, 533
 Möbius strip, 187
 parametric form, 131
 perfect specular, 508, 509
 properties, 508
 quadrics, 131
 retro-reflective, 508, 509
 smooth, 516
 subdivision, 181–206
- T**
- Tabulated BSSRDF, 696–700
- Tangents, 154, 155, 163
 across, 205
 computing for boundary vertices, 205–206
 computing for interior vertices, 204
 shading, 166
 transverse, 205–206
- Tetrahedron subdivision, 181
- Texel
 conversion, 621
 defined, 618
 representation, 619
 resampling, 621–625
- Texture(s), 597–664
 aliasing, 597, 598
 alpha mask, 154

- antialiasing, 598–608
 bilinear, 617–618
 bumpy, 660–662
 checkerboard, 436, 609, 642–646
 clamping, 605, 656, 666
 constant, 615
 displacement, 588, 589
 functions, 597
 functions, filtering, 604–605
 as high-frequency variation source, 597
 ideal resampling, 604
 image, 618–639
 implementations, 609
 interface, 614–618
 marble, 663–664
 mix, 616–617
 noise, 648–664
 parameters, 1119–1121
 polka dot, 653–655
 procedural, 640–648
 sampling, 598–608
 sampling rate, 599–604, 623
 scale, 615–616
 solid, 640–648
 solid checkerboard, 646–648
 supersampling, 605
 UV, 641–642
 windy waves, 662–663
 wrinkled, 660–662
- Texture coordinates, 608–614
 2D (u, v) mapping, 610–611
 3D mapping, 612–613
 checkerboard textures, 642
 cylindrical mapping, 612
 differential change, 614
 differentials for sphere, 611
 discontinuity, 612
 planar mapping, 613
 spherical mapping, 611–612
- Texture filtering
 elliptical weighted average, 634
 problem, 605
 for specular reflection, 605, 606
 triangular filter, 632
- Texture maps
 defined, 618
 filtering, 624
 procedural texturing and, 640
See also Image textures
- Texture projection lights, 724–728
 defined, 724
 directions behind, discarding, 727
 MIP map creation, 726
 projection matrix initialization, 726
 scene rendering, 725
- setting illustration, 725
See also Light(s); Point lights
- Texture space, 638
- Thick lens approximation, 386–388
- Thin lens approximation, 368
- Threading. *See* Parallelization
- Torrance-Sparrow model, 48, 544–549
 derivation of, 545, 548
 Fresnel term, 546
 geometric attenuation, 546
 half-angle vector, 545
 implementation, 546
 reflected outgoing radiance, 546
 terms evaluation, 547
See also Microfacet models
- Total internal reflection, 520
- Transactional memory, 42
- Transformations, 81–92
 affine, 104
 animating, 97–114
 applying, 93–97
 array, 1112
 benefits, 82
 of bounding boxes, 95
 composite matrices, 104
 composition of, 96, 104
 continuous, 82
 coordinate system handedness and, 96
 decomposition, 104
 defined, 82
 between distributions, 771–773
 of the frame, 82
 homogeneous coordinates, 82–84
 identity, 84
 invertible, 82
 keyframe, 97
 linear, 81
 look-at, 91–92
 matrix, 84, 96
 multidimensional, 773–784
 multiple dimensions, 772–773
 multiplying, 96
 of normals, 93–95
 from one frame to another, 82
 one-to-one, 82
 operations, 84–85
 of points, 93
 polar, 772
 primitive, 252–253
 of rays, 95
 rotation, 88–89
 scaling, 87–88
 scene description, 1111–1114
 translations, 85–87
 of vectors, 93
 world-to-object, 140
- Translations, 85–87
 in 2D, 86
 defined, 86
 extracting, 104
 interpolation, 98
 matrix equation, 86
 in matrix form, 85
 points and, 85
 properties, 85
- Translucent material, 584
- Transmission. *See* Specular transmission
- Transmittance, 13
 beam, 677–678
 in Equation of Transfer, 888–889
- Traversals
 bounding volume hierarchies (BVHs), 282–284
 kd-tree accelerator, 297–302
- Triangle filter
 defined, 477
 graph, 476
See also Filters; Reconstruction
- Triangle meshes
 coordinate assignment, 154
 defined, 152
 shape transformation, 155
 smooth-looking, 203
 tangent vectors, 163
See also Shapes
- Triangles, 152–167
 bound computation, 157
 dual role, 154
 intersection, 157–166
 object space bound, 157
 parametric coordinates, 163
 partial derivatives, 163, 164
 robust intersection, 221
 sampling, 781–782, 839–840
 as set of points, 163
 shading geometry, 167–168
 storage reduction, 156
 surface area, 167
 vertices, 156–157
 watertight intersection, 158, 221
 winding order, 165
 world space bounding boxes, 125
See also Shapes
- Tristimulus theory, 322
- Trowbridge-Reitz microfacet
 distribution function, 539
- Turbulence, 658–660
 defined, 658
 graphs of, 559
 implementation, 659
 noise octave computation, 660
See also Noise

U

Uber material, 584
 Unified resampling filters, 635
 Uniform scaling, 87
 Unit in last place (“ulp”), 209
 Unit vectors, 63, 65
 Utilities, 1061–1100
 communication with user, 1068–1070
 functions, 1062–1065
 image file input/output, 1066–1068
 main include file, 1061–1066
 mathematical routines, 1079–1081
 memory management, 1070–1079
 parallelism, 1082–1093
 pseudo-random number generator, 1065–1066
 statistics, 1093–1100
 UV texture, 641–642
 application illustration, 642
 defined, 641

V

Valence, 185
 Value noise, 649
 Van der Corput sequences, 443, 462, 463
 Variable stack allocation, 1070–1071
 Variance
 of functions, 750, 751
 Monte Carlo ray tracing, 747
 random sampling and, 791
 strata, 791
 sum of, 751
 Variance reduction
 bias, 793–794
 importance sampling, 794–799
 multiple importance sampling, 797–799
 sample placement, 789–793
 sampling reflection functions, 806–835
 Variation diminishing property (of splines), 170
 Varying-density volumes
 3D grids, 690–691
 Vector irradiance, 921
 Vectors, 59–67
 addition, 61
 basis, 57–58, 90
 coordinate system from, 67
 cross product, 64–65
 degenerate, 63
 difference, 62
 division, 62
 dot product, 63–65
 gradient, 651

half-angle, 545, 809
 homogeneous, 82–84
 multiplication, 62
 normal, 71–72, 93–95
 normalization, 65–66
 obtaining, 69
 orthogonal, 63
 orthonormal, 63
 rotation around arbitrary axis, 89
 scaling, 62
 specular reflection, 526, 527
 specular transmission, 527
 subtraction, 61–62
 tangent, 154, 155, 163, 166
 transforming, 93
 unary negation operator, 63
 unit, 63, 65
 “up”, 92
 Vertex
 abstraction layer, 995
 of light path, 867, 874, 891
 storage, 155
 of subdivision mesh, 185
 of triangle mesh, 152–153
 update rule for subdivision surfaces, 195
 Vertices (subdivision surface)
 boundary, determining, 190
 boundary, valence computation, 191
 child faces, 202
 edge computation, 199
 edges rule application, 199–200
 even, 194
 extraordinary, 185, 189, 194
 indices of, 189, 202
 initialization, 189, 190, 194, 199
 interior, 194
 limit surface position, 203
 local support, 195
 neighboring faces, 187
 new position computation, 194–200
 next face, 190, 191
 nonboundary, valence computation, 191
 numbering scheme, 200
 odd, 194, 201
 one-ring rule, 195, 204
 parent, 201
 previous face, 190
 regular, 185, 194
 sorting, 188
 split edges computation, 194
 tangents computation, 203
 updating values, 202
 weights, 195, 203
 Viewing volume, 5
 Vignetting, 31, 355
 Visibility, 10
 defined, 5
 Visibility testing, 717–719
 defined, 717
 Visible area sampling, 808, 810–811
 Volume light transport equation. *See*
 Equation of transfer
 Volume rendering, 887–895
 equation of transfer, 888–891
 subsurface scattering, 916–938
 Volume scattering, 671–704
 absorption, 672–674
 BSSRDF, 692–702
 emission, 672, 674–676
 extinction, 696–697, 896, 928
 heterogeneous, 672, 688–689, 893–894
 homogeneous, 672, 894–898
 illustrated, 672
 in-scattering, 678–680
 out-scattering, 676–678
 participating media, 671
 phase functions, 680–683, 898–899
 processes, 672–680
 properties, setting, 938–939
 Volumes
 light transport equation, 15

W

Warping samples, 792–793
 Watertight intersection, 158, 221, 243
 Web site, this book, 47
 Weingarten equations, 138
 Whitted integrator, 32–38
 geometric setting, 35
 radiance evaluation, 33
 scattered light and, 36
 Wien’s displacement law, 711
 Windowed sinc filter, 481–483
 Windy waves, 662–663
 defined, 662
 evaluation function, 662
 See also Noise
 World end, 1129–1131
 World space, 58
 bounding boxes, 125
 defined, 358
 primitive transformation to, 252
 rays, 135
 Wrinkled textures, 660–662

X

XYZ color, 322–325
 conversion from RGB, 328

- converting RGB to, 328
 - image film, 485
 - integrals, 332
 - matching curves, 322
 - SPD representation, 323
 - value computation, 323
 - values, 323
 - y coefficient, 325
- See also* Spectral representation

This page intentionally left blank

Physically Based Rendering

FROM THEORY TO IMPLEMENTATION

This book was typeset with \TeX , using the Zz \TeX macro package on the Microsoft Windows 7 platform. The main body of the text is set in Minion at 9.5/12, and the margin indices are set in Bitstream Letter Gothic 12 Pitch at 5.5/7. Chapter titles are set in East Bloc ICG Open and Univers Black. Cholla Sans Bold is used for other display headings.

The manuscript for this book was written in pyweb, a literate programming markup format of the authors' own design. This input format is based heavily on the noweb system developed by Norman Ramsey. The pyweb scripts simultaneously generate the \TeX files for the book as well as the source code of the pbrt system.

In addition, these scripts semi-automatically generate the code identifier cross-references that appear in the margin indices. Wherever possible, these indices are produced automatically by parsing the source code itself. Otherwise, usage and definition locations are marked explicitly in the pyweb input, and these special marks are removed before either the book or the code is generated. These scripts were originally written by the authors, and subsequently rewritten by Paul Anagnostopoulos in Gossip to integrate into the Zz \TeX package.

Overall, the book comprises over 80,000 lines of pyweb input, or nearly 3.5 megabytes of text. The cover image, example renderings, and chapter images were generated by pbrt, the software that is described in this book.

This page intentionally left blank

Index of Notation

ρ	Scattering albedo	676
σ_t	Attenuation (absorption) coefficient	676
$T_r(p \rightarrow p')$	Beam transmittance	677
$\tau(p \rightarrow p')$	Optical thickness	678
$p(\omega, \omega')$	Phase function	678
$L_s(p, \omega)$	Source term	680
$S_\omega(\omega_i)$	BSSRDF directional term	694
$S_p(p_i)$	BSSRDF profile term	694
$S_t(x)$	1D BSSRDF profile term	695
ρ_{eff}	Effective albedo	698
X, Y	Random variables	748
$P(x)$	Cumulative distribution function (CDF)	749
ξ	Canonical uniform random variable	749
$p(x)$	Probability density function (PDF)	750
$E_p[f(x)]$	Expected value of a function over distribution p	750
$E[f(x)]$	Expected value of a function over uniform distribution	750
$V[f(x)]$	Variance	750
$T(X \rightarrow X')$	Metropolis transition function	763
$a(X \rightarrow X')$	Metropolis acceptance probability	763
$p(y x)$	Conditional density function	774
J_T	Jacobian of multidimensional function	780
$\epsilon[F]$	Efficiency of Monte Carlo estimator	787
$t(p, \omega)$	Ray casting function	862
$L(p' \rightarrow p)$	Exitant radiance from p' to p	865
$f(p'' \rightarrow p' \rightarrow p)$	Three-point form of the BSDF	865
$G(p \leftrightarrow p')$	Geometric coupling term	866
$V(p \leftrightarrow p')$	Visibility term	866
$P(\bar{p}_i)$	Contribution of $i + 1$ vertex path	867
$T(\bar{p}_i)$	Throughput of $i + 1$ vertex path	868
β	Path throughput weight	876
$\hat{P}(\bar{p}_i)$	Generalized path contribution function	890
$\hat{T}(\bar{p}_i)$	Generalized path throughput function	891
$\hat{f}(p_{i+1} \rightarrow p_i \rightarrow p_{i-1})$	Generalized scattering distribution function	891
$\hat{G}(p \leftrightarrow p')$	Generalized geometric coupling term	891
σ'_s	Reduced scattering coefficient	918
σ'_t	Reduced attenuation coefficient	918
ρ'	Reduced albedo	918
$\phi(p)$	Fluence rate	921
$E(p)$	Vector irradiance	921
Q_i	i -th moment of medium emission	922
D	Classical diffusion coefficient	922
D_G	Grosjean's diffusion coefficient	924
$L_{ss}(p_1 \rightarrow p_0)$	Single-scattered radiance from a medium	930
$E_{ss}(p)$	Single-scattered irradiance at a point	931
W_e	Importance function	949
$f^*(\omega_o, \omega_i)$	Adjoint BSDF	960
$p^\rightarrow(x_i)$	Sampling probability for importance transport	1012
$p^\leftarrow(x_i)$	Sampling probability for radiance transport	1012
$N(x, \sigma)$	Normal distribution	1033

Matt Pharr, Wenzel Jakob, Greg Humphreys

PHYSICALLY BASED RENDERING

From Theory to Implementation

Third Edition

"Physically Based Rendering is a terrific book. It covers all the marvelous math, fascinating physics, practical software engineering, and clever tricks that are necessary to write a state-of-the-art photorealistic renderer. All of these topics are dealt with in a clear and pedagogical manner without omitting the all-important practical details."

- Per Christensen, Senior Software Developer, RenderMan Products, Pixar Animation Studios

Physically Based Rendering, Third Edition, describes both the mathematical theory behind modern photorealistic rendering as well as the practical implementation of a physically based rendering system. A method - known as 'literate programming' - combines human-readable documentation and source code into a single reference that is specifically designed to aid comprehension. With the ideas and software described in this book, you will learn how to design and implement a full-featured rendering system capable of creating stunning imagery.

This completely updated and revised edition includes:

- Substantial new coverage on bidirectional path tracing with multiple importance sampling, stochastic progressive photon mapping, Metropolis light transport, floating-point precision errors in ray tracing, subsurface scattering, realistic camera models, ray tracing hair and curves, microfacet reflection models, low-discrepancy sampling techniques, and much more.
- Extensively annotated C++ source code for a complete rendering system that implements the algorithms described in the book. The code is provided under a liberal open source license along with a repository of example scenes.

For the second edition of the book, the authors were awarded a Scientific and Technical Academy Award by the Academy of Motion Pictures Arts and Sciences for the impact the book has had on rendering in feature film production.

"This book has deservedly won an Academy Award. I believe it should also be nominated for a Pulitzer Prize."

- Donald Knuth, author of *The Art of Computer Programming*

Matt Pharr is a Software Engineer at Google and co-founder of both Neoptica (acquired by Intel) and Exluna (acquired by NVIDIA).

Wenzel Jakob is an assistant professor at EPFL's School of Computer and Communication Sciences.

Greg Humphreys is Director of Engineering at FanDuel, having previously worked at Google and NVIDIA. Before that, he was a professor of Computer Science at the University of Virginia.

PROGRAMMING

ISBN 978-0-12-800645-0



MORGAN KAUFMANN PUBLISHERS

AN IMPRINT OF ELSEVIER
elsevier.com