01 INTRODUCTION

Rendering is the process of producing a 2D image from a description of a 3D scene. Obviously, this is a very broad task, and there are many ways to approach it. *Physically based* techniques attempt to simulate reality; that is, they use principles of physics to model the interaction of light and matter. In physically based rendering, realism is usually the primary goal. This approach is in contrast to *interactive* rendering, which sacrifices realism for high performance and low latency, or *nonphotorealistic* rendering, which strives for artistic freedom and expressiveness.

This book describes `pbrt`, a physically based rendering system based on the ray-tracing algorithm. Most computer graphics books present algorithms and theory, sometimes combined with snippets of code. In contrast, this book couples the theory with a complete implementation of a fully functional rendering system. The source code to the system (as well as example scenes and a collection of data for rendering) can be found from the `pbrt` Web site's downloads page, *pbrt.org/downloads.php*.

1.1 LITERATE PROGRAMMING

While writing the TEX typesetting system, Donald Knuth developed a new programming methodology based on the simple but revolutionary idea that *programs should be written more for people's consumption than for computers' consumption*. He named this methodology *literate programming*. This book (including the chapter you're reading now) is a long literate program. This means that in the course of reading this book, you will read the *full* implementation of the `pbrt` rendering system, not just a high-level description of it.

Literate programs are written in a metalanguage that mixes a document formatting language (e.g., TEX or HTML) and a programming language (e.g., C++). Two separate systems process the program: a "weaver" that transforms the literate program into a document suitable for typesetting, and a "tangler" that produces source code suitable

for compilation. Our literate programming system is homegrown, but it was heavily influenced by Norman Ramsey's noweb system.

The literate programming metalanguage provides two important features. The first is the ability to mix prose with source code. This feature makes the description of the program just as important as its actual source code, encouraging careful design and documentation. Second, the language provides mechanisms for presenting the program code to the reader in an entirely different order than it is supplied to the compiler. Thus, the program can be described in a logical manner. Each named block of code is called a *fragment*, and each fragment can refer to other fragments by name.

As a simple example, consider a function `InitGlobals()` that is responsible for initializing all of a program's global variables:[1]

```
void InitGlobals(void) {
    num_marbles = 25.7;
    shoe_size = 13;
    dielectric = true;
    my_senator = REPUBLICAN;
}
```

Despite its brevity, this function is hard to understand without any context. Why, for example, can the variable `num_marbles` take on floating-point values? Just looking at the code, one would need to search through the entire program to see where each variable is declared and how it is used in order to understand its purpose and the meanings of its legal values. Although this structuring of the system is fine for a compiler, a human reader would much rather see the initialization code for each variable presented separately, near the code that actually declares and uses the variable.

In a literate program, one can instead write `InitGlobals()` like this:

⟨*Function Definitions*⟩ ≡
```
void InitGlobals() {
    ⟨Initialize Global Variables 2⟩
}
```

This defines a fragment, called ⟨*Function Definitions*⟩, that contains the definition of the `InitGlobals()` function. The `InitGlobals()` function itself refers to another fragment, ⟨*Initialize Global Variables*⟩. Because the initialization fragment has not yet been defined, we don't know anything about this function except that it will contain assignments to global variables. This is just the right level of abstraction for now, since no variables have been declared yet. When we introduce the global variable `shoe_size` somewhere later in the program, we can then write

⟨*Initialize Global Variables*⟩ ≡ 2
```
shoe_size = 13;
```

1 The example code in this section is merely illustrative and is not part of `pbrt` itself.

Here we have started to define the contents of ⟨*Initialize Global Variables*⟩. When the literate program is tangled into source code for compilation, the literate programming system will substitute the code `shoe_size = 13;` inside the definition of the `InitGlobals()` function. Later in the text, we may define another global variable, `dielectric`, and we can append its initialization to the fragment:

⟨*Initialize Global Variables*⟩ +≡ 2
```
dielectric = true;
```

The +≡ symbol after the fragment name shows that we have added to a previously defined fragment. When tangled, the result of these three fragments is the code

```
void InitGlobals() {
    shoe_size = 13;
    dielectric = true;
}
```

In this way, we can decompose complex functions into logically distinct parts, making them much easier to understand. For example, we can write a complicated function as a series of fragments:

⟨*Function Definitions*⟩ +≡
```
void complex_func(int x, int y, double *data) {
    ⟨Check validity of arguments⟩
    if (x < y) {
        ⟨Swap parameter values⟩
    }
    ⟨Do precomputation before loop⟩
    ⟨Loop through and update data array⟩
}
```

Again, the contents of each fragment are expanded inline in `complex_func()` for compilation. In the document, we can introduce each fragment and its implementation in turn. This decomposition lets us present code a few lines at a time, making it easier to understand. Another advantage of this style of programming is that by separating the function into logical fragments, each with a single and well-delineated purpose, each one can then be written and verified independently. In general, we will try to make each fragment less than 10 lines long.

In some sense, the literate programming system is just an enhanced macro substitution package tuned to the task of rearranging program source code. This may seem like a trivial change, but in fact literate programming is quite different from other ways of structuring software systems.

1.1.1 INDEXING AND CROSS-REFERENCING

The following features are designed to make the text easier to navigate. Indices in the page margins give page numbers where the functions, variables, and methods used on that page are defined. Indices at the end of the book collect all of these identifiers so that it's possible to find definitions by name. Appendix C, "Index of Fragments," lists the pages where each fragment is defined and the pages where it is used. Within the text, a defined

fragment name is followed by a list of page numbers on which that fragment is used. For example, a hypothetical fragment definition such as

⟨*A fascinating fragment*⟩ ≡ **184, 690**
```
num_marbles += .001;
```

indicates that this fragment is used on pages 184 and 690. If the fragments that use this fragment are not included in the book text, no page numbers will be listed. When a fragment is used inside another fragment, the page number on which it is first defined appears after the fragment name. For example,

⟨*Do something interesting*⟩+≡ **500**
```
InitializeSomethingInteresting();
⟨Do something else interesting 486⟩
CleanUp();
```

indicates that the ⟨*Do something else interesting*⟩ fragment is defined on page 486. If the definition of the fragment is not included in the book, no page number will be listed.

1.2 PHOTOREALISTIC RENDERING AND THE RAY-TRACING ALGORITHM

The goal of photorealistic rendering is to create an image of a 3D scene that is indistinguishable from a photograph of the same scene. Before we describe the rendering process, it is important to understand that in this context the word "indistinguishable" is imprecise because it involves a human observer, and different observers may perceive the same image differently. Although we will cover a few perceptual issues in this book, accounting for the precise characteristics of a given observer is a very difficult and largely unsolved problem. For the most part, we will be satisfied with an accurate simulation of the physics of light and its interaction with matter, relying on our understanding of display technology to present a good image to the viewer.

Most photorealistic rendering systems are based on the ray-tracing algorithm. Ray tracing is actually a very simple algorithm; it is based on following the path of a ray of light through a scene as it interacts with and bounces off objects in an environment. Although there are many ways to write a ray tracer, all such systems simulate at least the following objects and phenomena:

- *Cameras:* How and from where is the scene being viewed? Cameras generate rays from the viewing point into the scene.
- *Ray-object intersections:* We must be able to tell precisely where a given ray pierces a geometric object. In addition, we need to determine certain geometric properties of the object at the intersection point, such as a surface normal or its material. Most ray tracers also have some facility for finding the intersection of a ray with multiple objects, typically returning the closest intersection along the ray.
- *Light distribution:* Without lighting, there would be little point in rendering a scene. A ray tracer must model the distribution of light throughout the scene, including not only the locations of the lights themselves, but also the way in which they distribute their energy throughout space.

- *Visibility:* In order to know whether a given light deposits energy at a point on a surface, we must know whether there is an uninterrupted path from the point to the light source. Fortunately, this question is easy to answer in a ray tracer, since we can just construct the ray from the surface to the light, find the closest ray-object intersection, and compare the intersection distance to the light distance.
- *Surface scattering:* Each object must provide a description of its appearance, including information about how light interacts with the object's surface, as well as the nature of the reradiated (or *scattered*) light. We are usually interested in the properties of the light that is scattered directly toward the camera. Models for surface scattering are typically parameterized so that they can simulate a variety of appearances.
- *Recursive ray tracing:* Because light can arrive at a surface after bouncing off or passing through several other surfaces, it is usually necessary to trace additional rays originating at the surface to fully capture this effect. This is particularly important for shiny surfaces like metal or glass.
- *Ray propagation:* We need to know what happens to the light traveling along a ray as it passes through space. If we are rendering a scene in a vacuum, light energy remains constant along a ray. Although most human observers have never been in a vacuum, this is the typical assumption made by most ray tracers. More sophisticated models are available for tracing rays through fog, smoke, the Earth's atmosphere, and so on.

We will briefly discuss each of these simulation tasks in this section. In the next section, we will show `pbrt`'s high-level interface to the underlying simulation components and follow the progress of a single ray through the main rendering loop. We will also show one specific surface scattering model based on Turner Whitted's original ray-tracing algorithm.

1.2.1 CAMERAS

Nearly everyone has used a camera and is familiar with its basic functionality: you indicate your desire to record an image of the world (usually by pressing a button), and the image is recorded onto a piece of film or an electronic sensor. One of the simplest devices for taking photographs in the real world is called the *pinhole camera*. Pinhole cameras consist of a light-tight box with a tiny hole at one end (Figure 1.1). When the hole is uncovered, light enters this hole and falls on a piece of photographic paper that is affixed to the other end of the box. Despite its simplicity, this kind of camera is still used today, frequently for artistic purposes. Very long exposure times are necessary to get enough light on the film to form an image.

Although most cameras are substantially more complex than the pinhole camera, it is a convenient starting point for simulation. The most important function of the camera is to define the portion of the scene that will be recorded onto the film. In Figure 1.1, it is easy to see that connecting the pinhole to the edges of the film creates a double pyramid that extends into the scene. Objects that are not inside this pyramid cannot be imaged onto the film. Because modern cameras image a more complex shape than a pyramid, we will refer to the region of space that can potentially be imaged onto the film as the *viewing volume*.

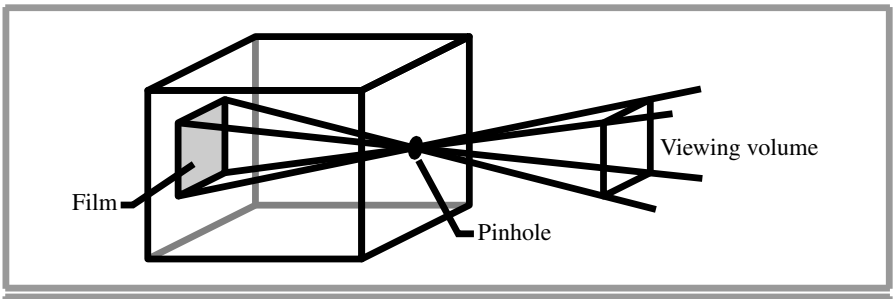

Figure 1.1: A Pinhole Camera.

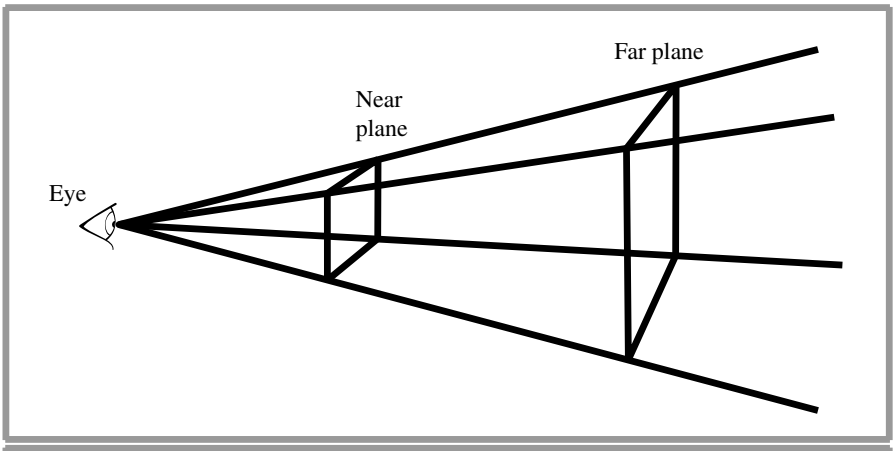

Figure 1.2: When we simulate a pinhole camera, we place the film in front of the hole at the near plane, and the hole is renamed the *eye*.

Another way to think about the pinhole camera is to place the film plane in *front* of the pinhole, but at the same distance (Figure 1.2). Note that connecting the hole to the film defines exactly the same viewing volume as before. Of course, this is not a practical way to build a real camera, but for simulation purposes it is a convenient abstraction. When the film (or image) plane is in front of the pinhole, the pinhole is frequently referred to as the *eye*.

Now we come to the crucial issue in rendering: at each point in the image, what color value do we display? If we recall the original pinhole camera, it is clear that only light rays that travel along the vector between the pinhole and a point on the film can contribute to that film location. In our simulated camera with the film plane in front of the eye, we are interested in the amount of light traveling from the image point to the eye.

Therefore, the task of the camera simulator is to take a point on the image and generate *rays* along which light is known to contribute to that image location. Because a ray consists of an origin point and a direction vector, this is particularly simple for the

pinhole camera model of Figure 1.2: it uses the pinhole for the origin, and the vector from the pinhole to the near plane as the ray's direction. For more complex camera models involving multiple lenses, the calculation of the ray that corresponds to a given point on the image may be more involved. However, if the process of converting image locations to rays is completely encapsulated in the camera module, the rest of the rendering system can focus on evaluating the lighting along those rays, and a variety of camera models can be supported. `pbrt`'s camera abstraction is described in detail in Chapter 6.

1.2.2 RAY-OBJECT INTERSECTIONS

Each time the camera generates a ray, the first task of the renderer is to determine which object, if any, that ray intersects first and where the intersection occurs. This intersection point is the visible point along the ray, and we will want to simulate the interaction of light with the object at this point. To find the intersection, we must test the ray for intersection against all objects in the scene and select the one that the ray intersects first. Given a ray r, we first start by writing it in *parametric form*:

$$\mathrm{r}(t) = \mathrm{o} + t\mathbf{d},$$

where o is the ray's origin, $\mathbf{d}$ is its direction vector, and t is a parameter whose legal range is $[0, \infty)$. We can obtain a point along the ray by specifying its parametric t value and evaluating the above equation.

It is often easy to find the intersection between the ray r and a surface defined by an implicit function $F(x, y, z) = 0$. We first substitute the ray equation into the implicit equation, producing a new function whose only parameter is t. We then solve this function for t and substitute the smallest positive root into the ray equation to find the desired point. For example, the implicit equation of a sphere centered at the origin with radius r is

$$x^2 + y^2 + z^2 - r^2 = 0,$$

so substituting the ray equation, we have

$$\left(\mathrm{o} + t\mathbf{d}\right)_x^2 + \left(\mathrm{o} + t\mathbf{d}\right)_y^2 + \left(\mathrm{o} + t\mathbf{d}\right)_z^2 - r^2 = 0.$$

This is just a quadratic equation in t, so we can easily solve it. If there are no real roots, the ray must miss the sphere. If there are roots, we select the smaller positive one to find the intersection point.

The intersection point is not enough information for the rest of the ray tracer; it needs to know certain properties of the surface at the point. First, the appearance model needs to be extracted and passed along to later stages of the ray-tracing algorithm, and additional geometric information about the intersection point will also be required in order to shade the point. For example, the surface normal $\mathbf{n}$ is always required. Although many ray tracers operate with only $\mathbf{n}$, more sophisticated rendering systems like `pbrt` require even more information, such as various partial derivatives of position and surface normal with respect to the local parameterization of the surface.

Of course, most scenes are made up of multiple objects. The brute-force intersection approach would be to test the ray against each object in turn, choosing the minimum t value of the intersections found. This approach, while correct, is very slow, even for

scenes of modest complexity. A solution is to incorporate an *acceleration structure* that quickly rejects whole groups of objects during the ray intersection process. This ability to quickly cull irrelevant geometry means that ray tracing frequently runs in $O(I \log N)$ time, where I is the number of pixels in the image and N is the number of objects in the scene.[2] (Building the acceleration structure is necessarily at least *O(N)* time.)

The geometric interface supported by `pbrt` is described in Chapter 3, and the acceleration interface is shown in Chapter 4.

1.2.3 LIGHT DISTRIBUTION

The ray-object intersection stage gives us a point to be shaded and some information about the geometry at that point. Recall that our eventual goal is to find the amount of light leaving this point in the direction of the camera. In order to do this, we need to know how much light is *arriving* at this point. This involves both the *geometric* and *radiometric* distribution of light in the scene. For very simple light sources (e.g., point lights), the geometric distribution of lighting is a simple matter of knowing the position of the lights. However, point lights do not exist in the real world, and so physically based lighting is often based on *area* light sources. This means that the light source is associated with a geometric object that emits illumination from its surface. However, we will use point lights in this section to illustrate the components of light distribution; rigorous discussion of light measurement and distribution is the topic of Chapters 5 and 12.

We frequently would like to know the amount of light energy being deposited on the differential area surrounding the intersection point (Figure 1.3). We will assume that the point light source has some power Φ associated with it, and that it radiates light equally in all directions. This means that the total amount of energy on a sphere surrounding the light is $\Phi/(4\pi)$. (These measurements will be explained and formalized in Chapter 12.) If we consider two such spheres (Figure 1.4), it is clear that the energy at a point on the larger sphere must be less than the energy at a point on the smaller sphere because the same total energy is distributed over a larger area. Specifically, the amount of energy arriving at a point on a sphere of radius r is proportional to $1/r^2$. Furthermore, it can be shown that if the tiny surface patch $\mathrm{d}A$ is tilted by an angle θ away from the vector from the surface point to the light, the amount of energy deposited on $\mathrm{d}A$ is proportional to $\cos\theta$. Putting this all together, the total light energy $\mathrm{d}E$ (the *differential irradiance*) deposited on $\mathrm{d}A$ is

$$\mathrm{d}E = \frac{\Phi \cos\theta}{4\pi r^2}.$$

Readers already familiar with basic lighting in computer graphics will notice two familiar laws encoded in this equation: the cosine falloff of light for tilted surfaces mentioned above, and the one-over-r-squared falloff of light with distance.

---

2 Although ray tracing's logarithmic complexity is often heralded as one of its key strengths, this is typically only true on average. The computational geometry literature has shown ray-tracing algorithms that have guaranteed logarithmic running time, but these algorithms only work for certain types of scenes and have very expensive preprocessing and storage requirements. Szirmay-Kalos and Márton provide pointers to the relevant literature (Szirmay-Kalos and Márton 1998). In practice, the ray intersection algorithms presented in this book are sublinear, but without expensive preprocessing and huge memory usage it is always possible to construct worst-case scenes where ray tracing runs in $O(IN)$ time.

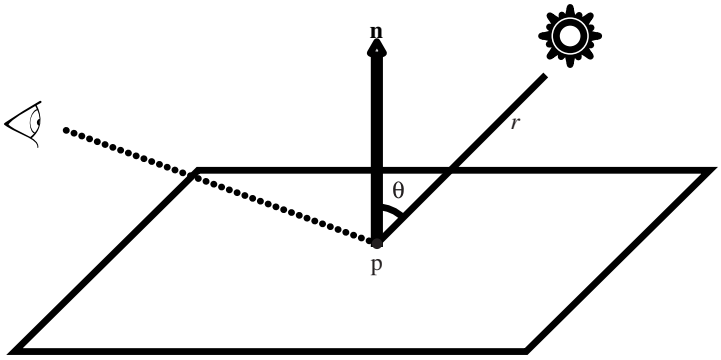

Figure 1.3: Geometric construction for evaluating the light energy at a point due to a point light source. The distance from the point to the light source is denoted by r.

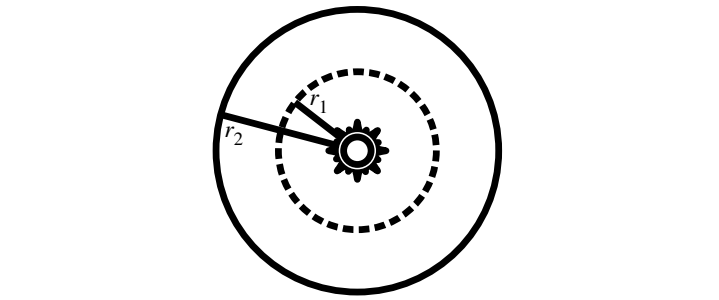

Figure 1.4: Since the point light radiates light equally in all directions, the same total energy is deposited on all spheres centered at the light.

Scenes with multiple lights are easily handled because illumination is *linear:* the contribution of each light can be computed separately and summed to obtain the overall contribution.

1.2.4 VISIBILITY

The lighting distribution described in the previous section ignores one very important component: *shadows*. Each light contributes illumination to the point being shaded only if the path from the point to the light's position is unobstructed (Figure 1.5).

Fortunately, in a ray tracer it is easy to determine if the light is visible from the point being shaded. We simply construct a new ray whose origin is at the surface point and whose direction points toward the light. These special rays are called *shadow rays*. If we trace this ray through the environment, we can check to see whether any intersections are found between the ray's origin and the light source by comparing the parametric t value of any

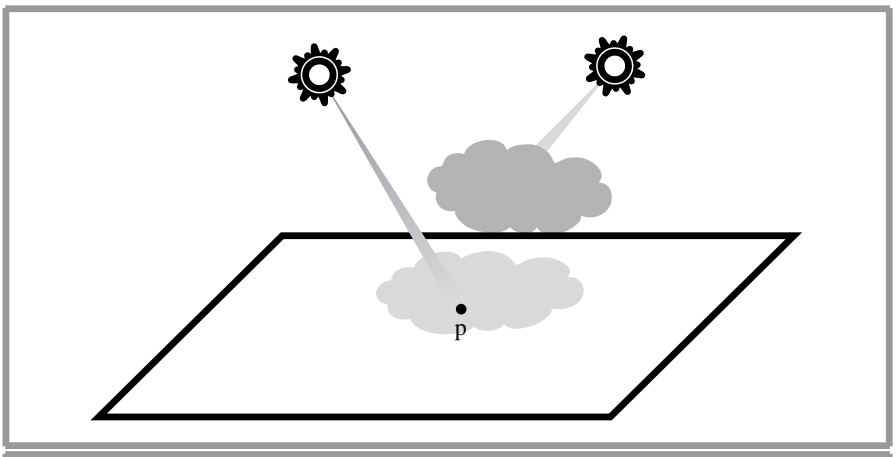

Figure 1.5: A light source only deposits energy on a surface if the source is not obscured as seen from the receiving point. The light source on the left illuminates the point p, but the light source on the right does not.

intersections found to the parameteric t value along the ray of the light source position. If there is no blocking object between the light and the surface, the light's contribution is included.

1.2.5 SURFACE SCATTERING

We now are able to compute two pieces of information that are vital for proper shading of a point: its location and the incident lighting.[3] Now we need to determine how the incident lighting is *scattered* at the surface. Specifically, we are interested in the amount of light energy scattered back along the ray that we originally traced to find the intersection point, since that ray leads to the camera (Figure 1.6).

Each object in the scene provides a *material*, which is a description of its appearance properties at each point on the surface. This description is given by the *Bidirectional Reflectance Distribution Function* (BRDF). This function tells us how much energy is reflected from a given incoming direction ω_i to a given outgoing direction ω_o. We will write the BRDF at p as $f_r(p, \omega_o, \omega_i)$. Now, computing the amount of light L scattered back toward the camera is straightforward:

```
for each light:
    if light is not blocked:
        incident_light = light.L(point)
        amount_reflected =
            surface.BRDF(hit_point, camera_vector, light_vector)
        L += amount_reflected * incident_light
```

3 Readers already familiar with rendering might object that the discussion in this section considers only direct lighting. Rest assured that `pbrt` does support global illumination.

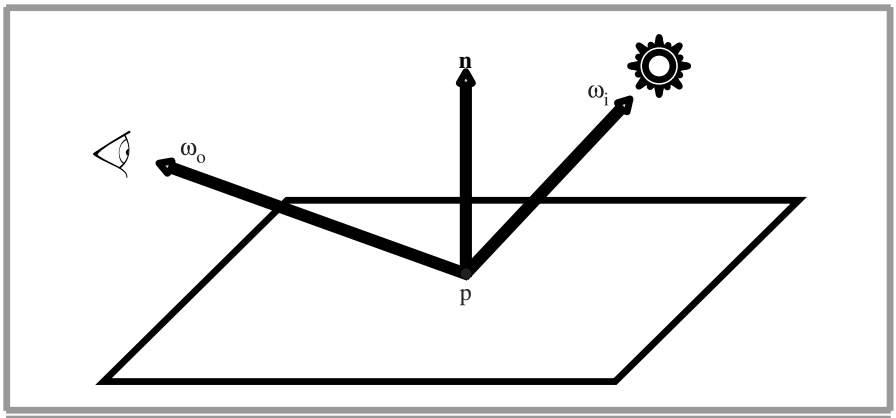

Figure 1.6: The Geometry of Surface Scattering. Incident light arriving along direction ω_i interacts with the surface at point p and is scattered back toward the camera along direction ω_o. The amount of light scattered toward the camera is given by the product of the incident light energy and the BRDF.

Here we are using L to represent the light; this represents a slightly different unit for light measurement than dE, which was used before.

It is easy to generalize the notion of a BRDF to transmitted light (obtaining a BTDF) or to general scattering of light arriving from either side of the surface. A function that describes general scattering is called a *Bidirectional Scattering Distribution Function* (BSDF). `pbrt` supports a variety of both physically and phenomenologically based BSDF models; they are described in Chapter 8.

1.2.6 RECURSIVE RAY TRACING

Turner Whitted's original paper on ray tracing emphasized its *recursive* nature. For example, if a ray from the camera hits a shiny object like a mirror, we can *reflect* the ray about the surface normal at the intersection point and recursively invoke the ray-tracing routine to find the light arriving at the point on the mirror, adding its contribution to the original camera ray. This same technique can be used to trace transmitted rays that intersect transparent objects. For a long time, most early ray-tracing examples showcased mirrors and glass balls (Figure 1.7) because these types of effects were difficult to capture with other rendering techniques.

In general, the amount of light that reaches the camera from a point on an object is given by the sum of light emitted by the object (if it is itself a light source) and the amount of reflected light. This idea is formalized by the *light transport equation* (also often known as the *rendering equation*), which says that the outgoing radiance $L_o(p, \omega_o)$ from a point p in direction ω_o is the emitted radiance at that point in that direction, $L_e(p, \omega_o)$, plus the incident radiance from all directions on the sphere $\mathcal{S}^2$ around p scaled by the BSDF $f(p, \omega_o, \omega_i)$ and a cosine term:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{\mathcal{S}^2} f(p, \omega_o, \omega_i)\, L_i(p, \omega_i)\, |\cos \theta_i|\, d\omega_i. \quad [1.1]$$

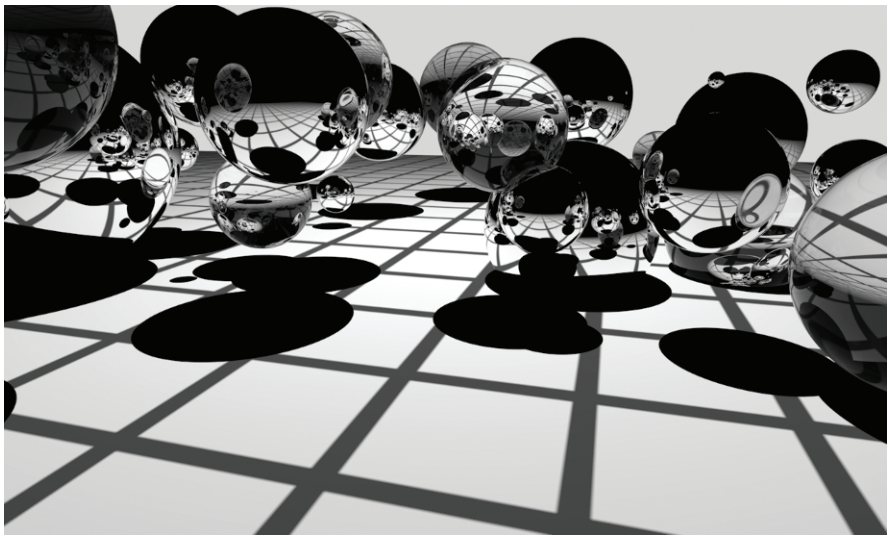

Figure 1.7: A Prototypical Example of Early Ray Tracing. Note the use of mirrored and glass objects, which emphasize the algorithm's ability to handle these kinds of surfaces.

We will show a more complete derivation of this equation in Sections 5.6.1 and 15.2. Solving this integral analytically is not possible except for the simplest of scenes, so we must either make simplifying assumptions or use numerical integration techniques.

Whitted's algorithm simplifies this integral by ignoring incoming light from most directions and only evaluating $L_i(p, \omega_i)$ for directions to light sources and for the directions of perfect reflection and refraction. In other words, it turns the integral into a sum over a small number of directions.

Whitted's method can be extended to capture more effects than just perfect mirrors and glass. For example, by tracing many recursive rays near the mirror-reflection direction and averaging their contributions, we obtain an approximation of glossy reflection. In fact, we can *always* recursively trace a ray whenever we hit an object. For example, we can randomly choose a reflection direction ω_i and weight the contribution of this newly spawned ray by evaluating the BRDF $f_r(p, \omega_o, \omega_i)$. This simple but powerful idea can lead to very realistic images because it captures all of the interreflection of light between objects. Of course, we need to know when to terminate the recursion, and choosing directions completely at random may make the rendering algorithm slow to converge to a reasonable result. These problems can be addressed, however; these issues are the topic of Chapters 13 to 15.

When we trace rays recursively in this manner, we are really associating a *tree* of rays with each image location (Figure 1.8), with the ray from the camera at the root of this tree. Note that each ray in this tree can have a *weight* associated with it; this allows us to model, for example, shiny surfaces that do not reflect 100% of the incoming light.

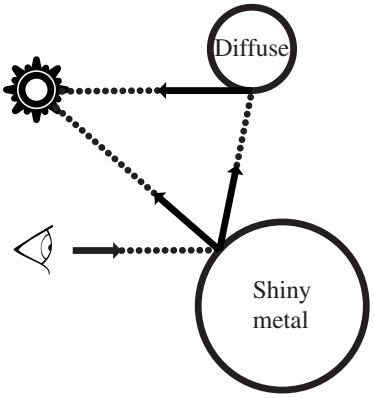

Figure 1.8: Recursive ray tracing associates an entire tree of rays with each image location.

1.2.7 RAY PROPAGATION

The prior discussion has assumed that rays are traveling through a vacuum. For example, when describing the distribution of light from a point source, we assumed that the energy was distributed equally on the surface of a sphere centered at the light without decreasing along the way. The presence of *participating media* such as smoke, fog, or dust can invalidate this assumption. Many ray tracers ignore these phenomena, though doing so is quite limiting. Even if we are not making a rendering of a smoke-filled room, almost all outdoor scenes are affected substantially by participating media. For example, Earth's atmosphere causes objects that are farther away to appear less saturated (Figure 1.9).

There are two ways in which a participating medium can affect the light propagating along a ray. First, the medium can *extinguish* (or *attenuate*) light, either by absorbing it or by scattering it in a different direction. We can capture this effect by computing the *transmittance* T between the ray origin and the intersection point. The transmittance tells us how much of the light scattered at the intersection point makes it back to the ray origin.

A participating medium can also add to the light along a ray. This can happen either if the medium emits light (as with a flame) or if the medium scatters light from other directions back along the ray (Figure 1.10). We can find this quantity by numerically evaluating the *volume light transport equation*, in the same way we evaluated the light transport equation to find the amount of light reflected from a surface. We will leave the description of participating media and volume rendering until Chapters 11 and 16. For now, it will suffice to say that we can compute the effect of participating media and incorporate its effect into the amount of light carried by the ray.

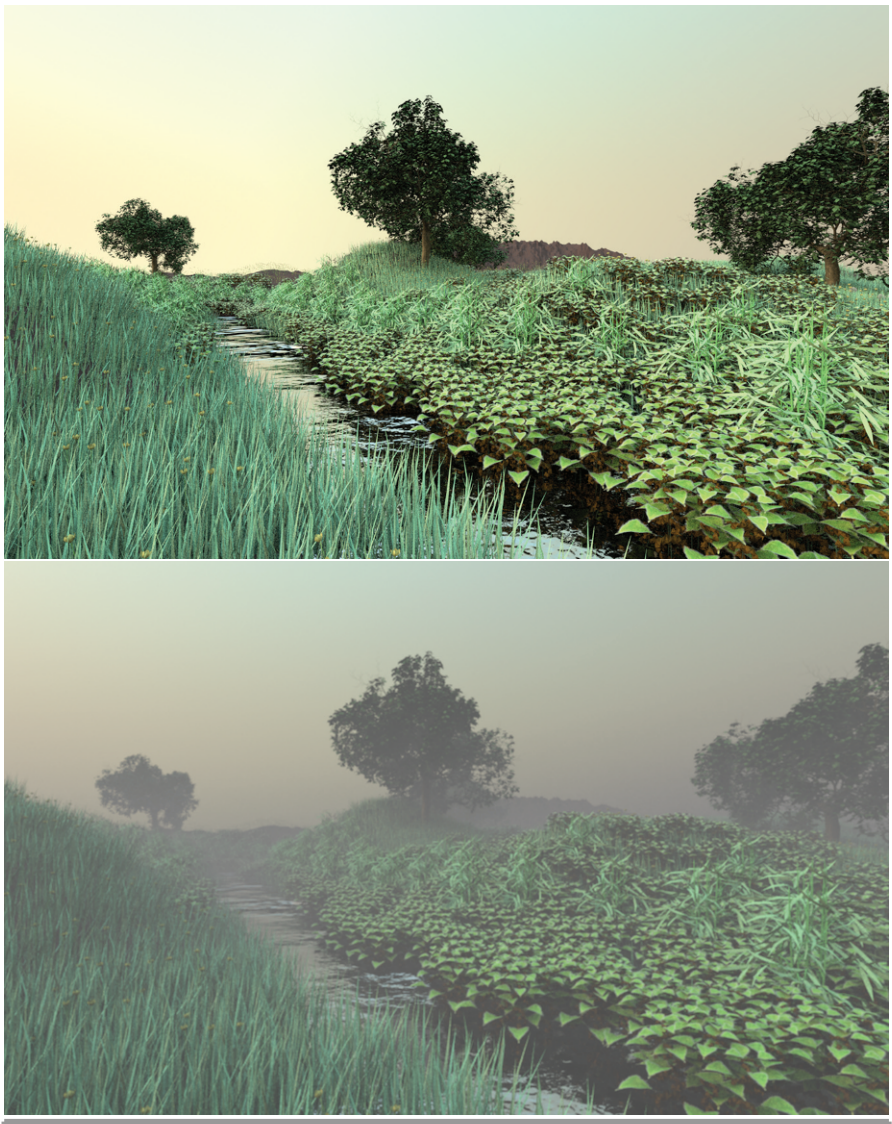

Figure 1.9: Earth's Atmosphere Decreases Saturation with Distance. The scene on the top is rendered without simulating this phenomenon, while the scene on the bottom includes an atmospheric model. This sort of atmospheric attenuation is an important depth cue when viewing real scenes and adds a sense of scale to the rendering on the bottom.

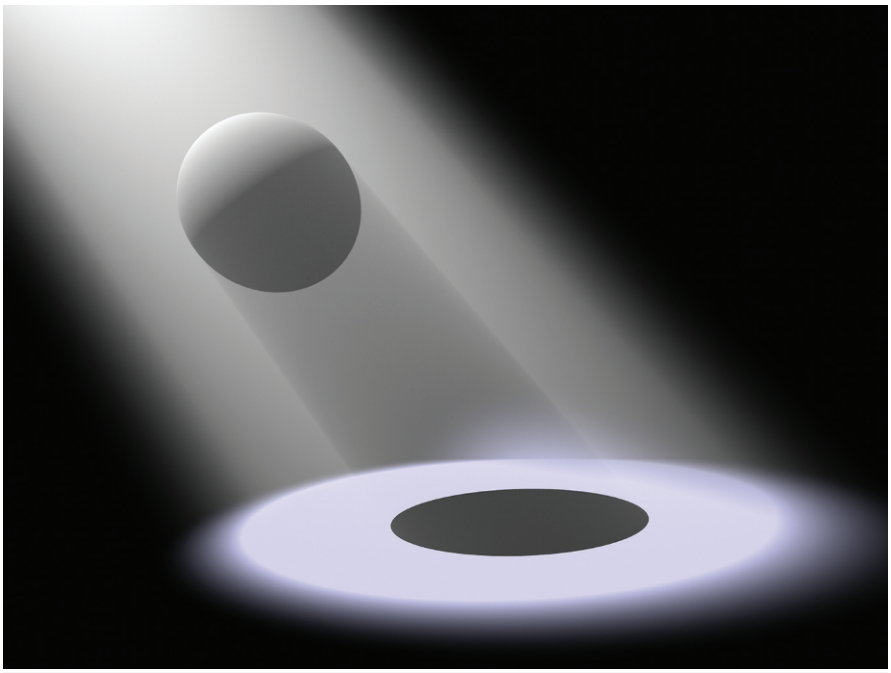

Figure 1.10: A Spotlight Shining on a Sphere through Fog. Notice that the shape of the spotlight's lighting distribution and the sphere's shadow are clearly visible due to the additional scattering in the participating medium.

1.3 pbrt: SYSTEM OVERVIEW

`pbrt` is structured using standard object-oriented techniques: abstract base classes are defined for important entities (for example, a `Shape` abstract base class defines the interface that all geometric shapes must implement, the `Light` abstract base class acts similarly for lights, etc.). The majority of the system is implemented purely in terms of the interfaces provided by these abstract base classes; for example, the code that checks for occluding objects between a light source and a point being shaded calls the `Shape` intersection methods and doesn't need to consider the particular types of shapes that are present in the scene. This approach makes it easy to extend the system, as adding a new shape only requires implementing a class that implements the `Shape` interface and linking it into the system.

`pbrt` is written using a total of 13 key abstract base classes, summarized in Table 1.1. Adding a new implementation of one of these types to the system is straightforward; the implementation must inherit from the appropriate base class, be compiled and linked into the executable, and the object creation routines in Appendix B must be modified to create instances of the object as needed as the scene description file is parsed. Section B.4 discusses extending the system in this manner in more detail.

Table 1.1: Main Interface Types. Most of pbrt is implemented in terms of 13 key abstract base classes, listed here. Implementations of each of these can easily be added to the system to extend its functionality.

Base class	Directory	Section
Shape	shapes/	3.1
Aggregate	accelerators/	4.2
Camera	cameras/	6.1
Sampler	samplers/	7.2
Filter	filters/	7.7
Film	film/	7.8
Material	materials/	9.2
Texture	textures/	10.3
VolumeRegion	volumes/	11.3
Light	lights/	12.1
Renderer	renderers/	1.3.3
SurfaceIntegrator	integrators/	Ch. 15 intro
VolumeIntegrator	integrators/	16.2

The source code to pbrt is distributed across a small directory hierarchy that can be found in the pbrt distribution, available from *pbrt.org/downloads.php*. All of the code for the pbrt core is in the src/core/ directory, and the main() function is contained in the short file src/main/pbrt.cpp. Various implementations of instances of the abstract base classes are in separate directories: src/shapes/ has implementations of the Shape base class, src/materials/ has implementations of Material, and so forth.

Throughout this section are a number of images rendered with extended versions of pbrt. Of them, Figures 1.11 through 1.14 are notable: not only are they visually impressive, but each of them was created by a student in a rendering course where the final class project was to extend pbrt with new functionality in order to render an interesting image. These images are among the best from those courses. Figures 1.15 and 1.16 were rendered with *LuxRender*, a GPL-licensed rendering system originally based on the pbrt source code from the first edition of the book. (See *www.luxrender.net* for more information about *LuxRender*.)

Aggregate 192
Camera 302
Film 403
Filter 393
Light 606
main() 20
Material 483
Renderer 24
Sampler 340
Shape 108
SurfaceIntegrator 740
Texture 519
VolumeIntegrator 876
VolumeRegion 587

1.3.1 PHASES OF EXECUTION

pbrt can be conceptually divided into two phases of execution. First, it parses the scene description file provided by the user. The scene description is a text file that specifies the geometric shapes that make up the scene, their material properties, the lights that illuminate them, where the virtual camera is positioned in the scene, and parameters to all of the individual algorithms used throughout the system. Each statement in the input file has a direct mapping to one of the routines in Appendix B; these routines comprise the procedural interface for describing a scene to pbrt. A number of example scenes are provided in the scenes/ directory in the pbrt distribution. The scene file format is

Figure 1.11: Guillaume Poncin and Pramod Sharma extended `pbrt` in numerous ways, implementing a number of complex rendering algorithms, to make this prize-winning image for Stanford's cs348b rendering competition. The trees are modeled procedurally with L-systems, a glow image processing filter increases the apparent realism of the lights on the tree, snow was modeled procedurally with metaballs, and a subsurface scattering algorithm gave the snow its realistic appearance by accounting for the effect of light that travels beneath the snow for some distance before leaving it.

specified in the file `docs/fileformat.pdf` and a user's guide to the file format is in the file `docs/usersguide.pdf`.

The end results of the parsing phase are an instance of the `Scene` class and an instance of the `Renderer` class. The `Scene` specifies the contents of the scene (geometric objects, lights, etc.), and the `Renderer` implements an algorithm to render it.

Once the scene has been specified, the second phase of execution begins and the main rendering loop executes. This phase is where `pbrt` usually spends the majority of its running time, and most of this book describes code that executes during this phase. The rendering loop is implemented in an implementation of the `Renderer::Render()` method, which is the focus of Section 1.3.4.

For the `SamplerRenderer` described in this chapter, the `Render()` method determines the light arriving at a virtual film plane for a large number of rays in order to model the process of image formation. After the contributions of all of these film samples have been computed, the final image is written to disk. The scene description data in memory are deallocated and the renderer then resumes processing statements from the scene description file until no more remain, allowing the user to specify another scene to be rendered, if desired.

`Renderer` 24
`Renderer::Render()` 24
`SamplerRenderer` 25
`Scene` 22

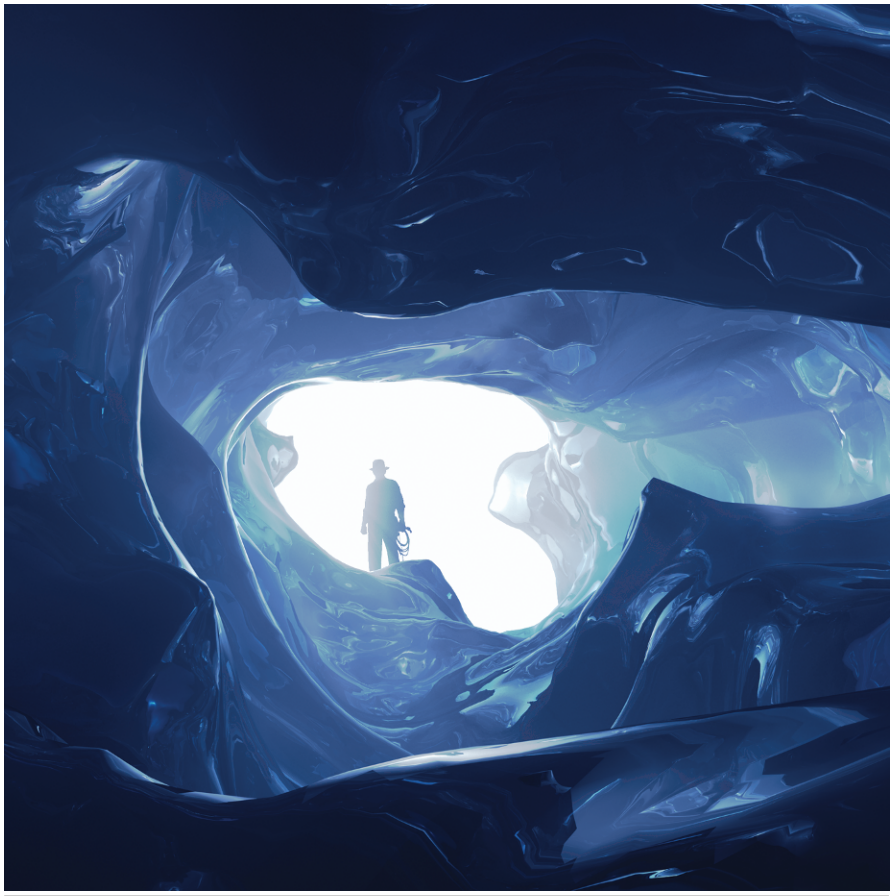

Figure 1.12: Abe Davis, David Jacobs, and Jongmin Baek rendered this amazing image of an ice cave to take the grand prize in the 2009 Stanford CS348b rendering competition. They first implemented a simulation of the physical process of glaciation, the process where snow falls, melts, and refreezes over the course of many years, forming stratified layers of ice. They then simulated erosion of the ice due to melted water runoff before generating a geometric model of the ice. Scattering of light inside the volume was simulated with volumetric photon mapping; the blue color of the ice is entirely due to modeling the wavelength-dependent absorption of light in the ice volume.

1.3.2 SCENE REPRESENTATION

`pbrt`'s `main()` function can be found in the file `main/pbrt.cpp`. This function is quite simple; it first loops over the provided command-line arguments in `argv`, initializing values in the `Options` structure and storing the filenames provided in the arguments. Running `pbrt` with `--help` as a command line argument prints all of the options that can be specified on the command line. The fragment that parses the command-line arguments, ⟨*Process command-line arguments*⟩, is straightforward and therefore not included in the book here.

`main()` 20

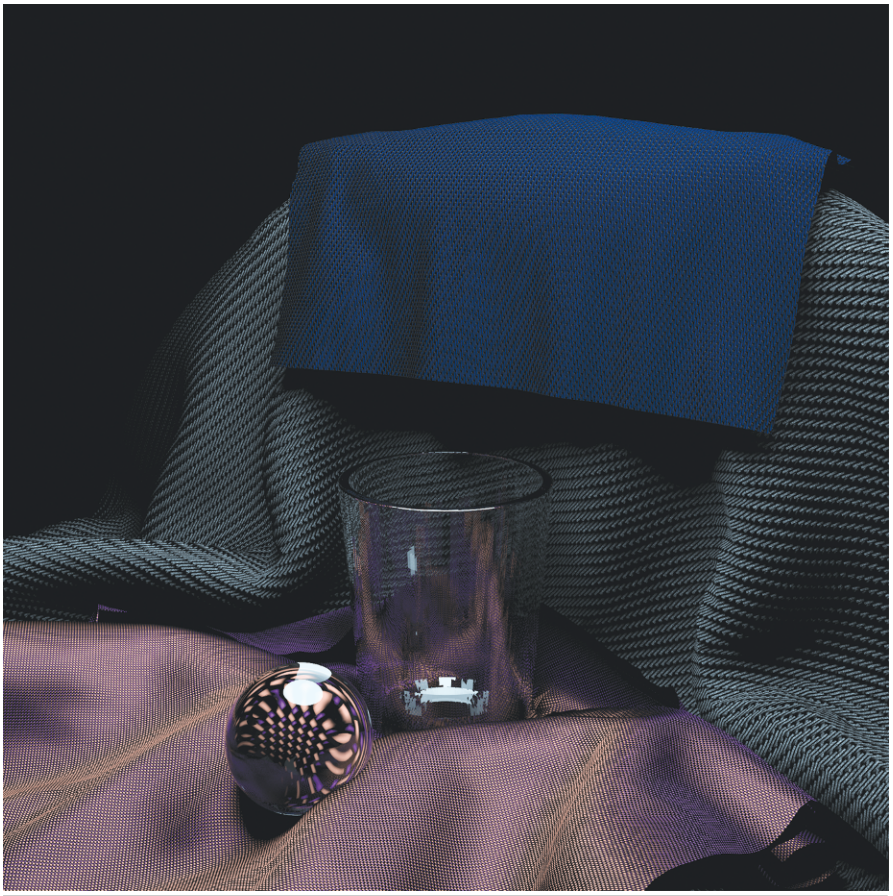

Figure 1.13: Lingfeng Yang implemented a bidirectional texture function to simulate the appearance of cloth, adding an analytic self-shadowing model, to render this image that took first prize in the 2009 Stanford CS348b rendering competition.

The options structure is then passed `pbrtInit()`, which does systemwide initialization. The `main()` function then parses the given scene description(s), leading to the creation of a `Scene` object that represents all of the elements (shapes, lights, etc.) that make up the scene and a `Renderer` object that implements an algorithm to render the scene. Because the input file can specify multiple scenes to be rendered, rendering actually begins as soon as the appropriate input directive is parsed. After all rendering is done, `pbrtCleanup()` does final cleanup before the system exits.

`pbrtCleanup()` 1052
`pbrtInit()` 1051
`Renderer` 24
`Scene` 22

The `pbrtInit()` and `pbrtCleanup()` functions appear in a *mini-index* in the page margin, along with the number of the page where they are actually defined. The mini-indices have pointers to the definitions of almost all of the functions, classes, methods, and member variables used or referred to on each page.

Figure 1.14: Jared Jacobs and Michael Turitzin added an implementation of Kajiya and Kay's texel-based fur rendering algorithm (Kajiya and Kay 1989) to `pbrt` and rendered this image, where both the fur on the dog and the shag carpet are rendered with the texel fur algorithm.

⟨*main program*⟩ ≡

```
int main(int argc, char *argv[]) {
    Options options;
    vector<string> filenames;
    ⟨Process command-line arguments⟩
    pbrtInit(options);
    ⟨Process scene description 21⟩
    pbrtCleanup();
    return 0;
}
```

Options 1051
pbrtCleanup() 1052
pbrtInit() 1051

If `pbrt` is run with no input filenames provided, then the scene description is read from standard input. Otherwise it loops through the provided filenames, processing each file in turn.

Figure 1.15: This contemporary indoor scene was modeled and rendered by Florent Boyer (*www.florentboyer.com*). The image was rendered using *LuxRender*, a GPL licensed physically-based rendering system originally based on `pbrt`'s source code. Modelling and texturing was done using Blender.

⟨*Process scene description*⟩ ≡ 20
```
if (filenames.size() == 0) {
    ⟨Parse scene from standard input 21⟩
} else {
    ⟨Parse scene from input files 21⟩
}
```

The `ParseFile()` function parses a scene description file, either from standard input or from a file on disk; it returns `false` if it was unable to open the file. The mechanics of parsing scene description files will not be described in this book; the parser implementation can be found in the `lex` and `yacc` files `core/pbrtlex.ll` and `core/pbrtparse.yy`, respectively. Readers who want to understand the parsing subsystem but are not familiar with these tools may wish to consult Levine, Mason, and Brown (1992). We use the common UNIX idiom that a file named "`-`" represents standard input:

⟨*Parse scene from standard input*⟩ ≡ 21
```
ParseFile("-");
```

If a particular input file can't be opened, the `Error()` routine reports this information to the user. `Error()` uses the same format string semantics as `printf()`.

⟨*Parse scene from input files*⟩ ≡ 21
```
for (u_int i = 0; i < filenames.size(); i++)
    if (!ParseFile(filenames[i]))
        Error("Couldn't open scene file \"%s\"", filenames[i].c_str());
```

`Error()` 1005
`ParseFile()` 21

Figure 1.16: Martin Lubich modeled this scene of the Austrian Imperial Crown and rendered it using *LuxRender*, an open-source fork of the `pbrt` codebase. The scene was modeled in Blender and consists of approximately 1.8 million vertices. It is illuminated by six area light sources with emission spectra based on measured data from a real-world light source and was rendered with 1280 samples per pixel in 73 hours of computation on a quad-core CPU. See Martin's Web site, *www.loramel.net*, for more information, including downloadable Blender scene files.

As the scene file is parsed, objects are created that represent the lights and geometric primitives in the scene. These are all stored in the `Scene` object, which is created by the `RenderOptions::MakeScene()` method in Section B.3.7 in Appendix B. The `Scene` class is declared in `core/scene.h` and defined in `core/scene.cpp`.

We will not include the implementation of the `Scene` constructor here; it just stores copies of its arguments in the various member variables inside the class.

⟨*Scene Declarations*⟩ ≡

```
class Scene {
public:
    ⟨Scene Public Methods 23⟩
    ⟨Scene Public Data 23⟩
};
```

Each geometric object in the scene is represented by a `Primitive`, which combines two objects: a `Shape` that specifies its geometry, and a `Material` that describes its appearance (e.g., the object's color, whether it has a dull or glossy finish). All of these geometric

Material 483
Primitive 185
RenderOptions::
MakeScene() 1072
Scene 22
Shape 108

primitives are collected into a single aggregate `Primitive` in the `Scene` member variable `Scene::aggregate`. This aggregate is a special kind of primitive that itself holds references to many other primitives. Because it implements the `Primitive` interface it appears no different than a single primitive to the rest of the system. The specific class used to implement `Scene::aggregate` stores all the scene's primitives in an acceleration data structure that reduces the number of unnecessary ray intersection tests with primitives that a given ray doesn't pass near.

⟨*Scene Public Data*⟩ ≡ 22
```
Primitive *aggregate;
```

Each light source in the scene is represented by a `Light` object, which specifies the shape of a light and the distribution of energy that it emits. The `Scene` stores all of the lights in a `vector` class from the C++ standard library. While some renderers support separate light lists per geometric object, allowing a light to illuminate only some of the objects in the scene, this idea does not map well to the physically based rendering approach taken in `pbrt`, so we use only this per-scene list.

⟨*Scene Public Data*⟩ +≡ 22
```
vector<Light *> lights;
```

In addition to geometric primitives, `pbrt` also supports participating media, or *volumetric* primitives. These types of primitives are supported through the `VolumeRegion` interface. The system's support for participating media is described in Chapter 11. Like `Primitives`, multiple `VolumeRegions` are all stored together in a single aggregate region, `Scene::volumeRegion`.

⟨*Scene Public Data*⟩ +≡ 22
```
VolumeRegion *volumeRegion;
```

The `Scene` class provides a handful of additional methods. Its `Intersect()` method traces the given ray into the scene and returns a Boolean value indicating whether the ray intersected any of the primitives. If so, it fills in the provided `Intersection` structure with information about the closest intersection point along the ray. The `Intersection` structure is defined in Section 4.1.

Intersection 186
Light 606
Primitive 185
Primitive::Intersect() 186
Ray 66
Scene 22
Scene::aggregate 23
Scene::Intersect() 23
Scene::IntersectP() 24
Scene::volumeRegion 23
VolumeRegion 587

⟨*Scene Public Methods*⟩ ≡ 22
```
bool Intersect(const Ray &ray, Intersection *isect) const {
    bool hit = aggregate->Intersect(ray, isect);
    return hit;
}
```

A closely related method is `Scene::IntersectP()`, which checks for the existence of intersections along the ray, but does not return any information about those intersections. Because this routine doesn't need to search for the closest intersection or compute any additional information about the intersections, it is generally more efficient than `Scene::Intersect()`. This routine is used for shadow rays.

⟨*Scene Public Methods*⟩ +≡ 22
```
bool IntersectP(const Ray &ray) const {
    bool hit = aggregate->IntersectP(ray);
    return hit;
}
```

Finally, `Scene::WorldBound()` returns a 3D box that bounds all of the geometry in the scene, which is simply the bounding box of `Scene::aggregate`. The `Scene` class caches this bound to avoid having to repeatedly compute it.

⟨*Scene Constructor Implementation*⟩ ≡
```
bound = aggregate->WorldBound();
if (volumeRegion) bound = Union(bound, volumeRegion->WorldBound());
```

⟨*Scene Public Data*⟩ +≡ 22
```
BBox bound;
```

⟨*Scene Method Definitions*⟩ ≡
```
const BBox &Scene::WorldBound() const {
    return bound;
}
```

1.3.3 RENDERER INTERFACE AND `SamplerRenderer`

Rendering an image of the scene is handled by an instance of a class that implements the `Renderer` interface. `Renderer` is an abstract base class that defines a few methods that must be provided by all renderers. It is defined in the files `core/renderer.h` and `core/renderer.cpp`. In this section, we will define both the interface that all `Renderers` must provide as well as the start of one instance of a `Renderer`, the `SamplerRenderer`.

⟨*Renderer Declarations*⟩ ≡
```
class Renderer {
public:
    ⟨Renderer Interface 24⟩
};
```

The main method that `Renderers` must provide is `Render()`; the `Renderer` is passed a pointer to a `Scene` and computes an image of the scene or more generally, a set of measurements of the scene lighting. For example, in Section 17.3, we define a completely different kind of `Renderer` that computes measurements of incident illumination at a set of points in the scene and writes the results to a text file, without generating an image at all. These measurements can then be used for interactive rendering of the scene, among other applications.

⟨*Renderer Interface*⟩ ≡ 24
```
virtual void Render(const Scene *scene) = 0;
```

`Renderers` are also required to provide methods that compute information about the illumination along rays in the scene. `Li()` returns the incident radiance along the given ray. In addition to the ray and the scene, it takes a number of additional parameters: the `Sample` (which may be `NULL`) provides random sample values for Monte Carlo integration

`BBox` 70
`Primitive::IntersectP()` 186
`Primitive::WorldBound()` 185
`Ray` 66
`Renderer` 24
`Sample` 343
`SamplerRenderer` 25
`Scene` 22
`Scene::aggregate` 23
`Scene::bound` 24
`Scene::volumeRegion` 23
`Scene::WorldBound()` 24
`Union()` 72

computations in the integrator, and the RNG is a pseudo-random number generator that is also available for this purpose. The MemoryArena performs efficient allocation of small temporary amounts of memory that may be needed while computing radiance along the ray. Finally, information about the ray's geometric intersection point can be returned via the Intersection, if its pointer is non-NULL, and the volumetric transmittance along the ray is returned via the T parameter, also if non-NULL.

⟨*Renderer Interface*⟩ +≡ 24
```
virtual Spectrum Li(const Scene *scene, const RayDifferential &ray,
    const Sample *sample, RNG &rng, MemoryArena &arena,
    Intersection *isect = NULL, Spectrum *T = NULL) const = 0;
```

Transmittance() returns the fraction of light that is attenuated by volumetric scattering along the ray. Renderer implementations will generally dispatch to Integrators (defined in Chapters 15 and 16) to compute the values returned by Li() and Transmittance().

⟨*Renderer Interface*⟩ +≡ 24
```
virtual Spectrum Transmittance(const Scene *scene,
    const RayDifferential &ray, const Sample *sample,
    RNG &rng, MemoryArena &arena) const = 0;
```

Now we will define the SamplerRenderer implementation of the Renderer interface and show how its Render() method computes an image of the scene. SamplerRenderer is so named because its rendering process is driven by a stream of *samples* from a Sampler; each such sample identifies a point on the image at which to compute the arriving light to form the image. The definition of SamplerRenderer is in the files renderers/samplerrenderer.h and renderers/samplerrenderer.cpp.

⟨*SamplerRenderer Declarations*⟩ ≡
```
class SamplerRenderer : public Renderer {
public:
    ⟨SamplerRenderer Public Methods⟩
private:
    ⟨SamplerRenderer Private Data 25⟩
};
```

Camera 302
Film 403
Integrator 740
Intersection 186
MemoryArena 1015
RayDifferential 69
Renderer 24
RNG 1003
Sample 343
Sampler 340
SamplerRenderer 25
Scene 22
Spectrum 263

The SamplerRenderer stores a pointer to a Sampler. The role of this class is subtle, but its implementation can substantially affect the quality of the images that the system generates. First, the sampler is responsible for choosing the points on the image plane from which rays are traced. Second, it is responsible for supplying the sample positions used by the integrators in their light transport computations; for example, some integrators need to choose random points on light sources to compute illumination from area lights. Generating a good distribution of these samples is an important part of the rendering process that can substantially affect overall efficiency and is discussed in Chapter 7.

⟨*SamplerRenderer Private Data*⟩ ≡ 25
```
Sampler *sampler;
```

The Camera object controls the viewing and lens parameters such as position, orientation, focus, and field of view. A Film member variable inside the Camera class handles image storage. The Camera classes are described in Chapter 6, and Film is described in

Section 7.8. The Film is responsible for writing the final image to disk and possibly displaying it on the screen as it is being computed.

⟨*SamplerRenderer Private Data*⟩ +≡ 25
```
Camera *camera;
```

Integrators handle the task of simulating the propagation of light in the scene in order to compute how much light arrives at image sample positions on the film plane. They are so named because they numerically evaluate the integrals in the surface and volume light transport equations that describe the distribution of light in the environment. SurfaceIntegrators compute reflected light from geometric surfaces, while VolumeIntegrators handle the scattering from volumetric primitives. Integrators are described in Chapters 15 and 16.

⟨*SamplerRenderer Private Data*⟩ +≡ 25
```
SurfaceIntegrator *surfaceIntegrator;
VolumeIntegrator *volumeIntegrator;
```

The SamplerRenderer constructor just stores pointers to these objects in member variables. The SamplerRenderer is created in the RenderOptions::MakeRenderer() method, which is in turn called by pbrtWorldEnd(), which is called by the input file parser when it is done parsing a scene description from an input file and is ready to render the scene.

⟨*SamplerRenderer Method Definitions*⟩ ≡
```
SamplerRenderer::SamplerRenderer(Sampler *s, Camera *c,
        SurfaceIntegrator *si, VolumeIntegrator *vi) {
    sampler = s;
    camera = c;
    surfaceIntegrator = si;
    volumeIntegrator = vi;
}
```

Camera 302
Film 403
pbrtWorldEnd() 1071
Renderer 24
Renderer::Render() 24
RenderOptions::MakeRenderer() 1072
Sampler 340
SamplerRenderer 25
SamplerRenderer::camera 26
SamplerRenderer::sampler 25
SamplerRenderer::surfaceIntegrator 26
SamplerRenderer::volumeIntegrator 26
Scene 22
SurfaceIntegrator 740
VolumeIntegrator 876

1.3.4 THE MAIN RENDERING LOOP

After the Scene and the Renderer have been allocated and initialized, the Renderer::Render() method is invoked, starting the second phase of pbrt's execution: the main rendering loop. In the SamplerRenderer's implementation of this method, at each of a series of positions on the image plane, the method uses the Camera and the Sampler to generate a ray into the scene, and then uses its integrators to determine the amount of light arriving at the image plane along that ray. This value is passed to the Film, which records the light's contribution. Figure 1.17 summarizes the main classes used in this method and the flow of data among them.

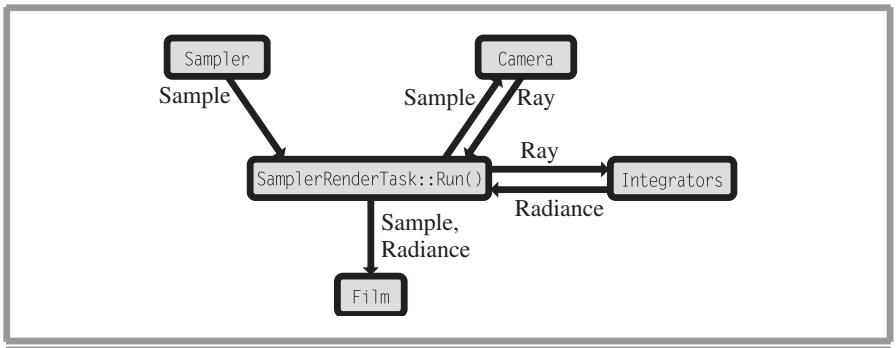

Figure 1.17: Class Relationships for the Main Rendering Loop in the `SamplerRenderer::Render()` Method in `renderers/sample.cpp`. The `Sampler` provides a sequence of sample values, one for each image sample to be taken. The `Camera` turns a sample into a corresponding ray from the film plane, and the `Integrators` compute the radiance along that ray arriving at the film. The sample and its radiance are given to the `Film`, which stores their contribution in an image. This process repeats until the `Sampler` has provided as many samples as are necessary to generate the final image.

⟨*SamplerRenderer Method Definitions*⟩ +≡
```
void SamplerRenderer::Render(const Scene *scene) {
    ⟨Allow integrators to do preprocessing for the scene 27⟩
    ⟨Allocate and initialize sample 27⟩
    ⟨Create and launch SamplerRendererTasks for rendering image 28⟩
    ⟨Clean up after rendering and store final image 29⟩
}
```

Before rendering can begin, the `SamplerRenderer` calls the `Preprocess()` methods of the integrators. Because information like the specific light sources and geometry of the scene aren't available when the integrators are first created, the `Preprocess()` method gives them an opportunity to do any necessary scene-dependent initialization or preprocessing. For example, the `PhotonIntegrator` in Section 15.6 uses this opportunity to create data structures that hold a representation of the distribution of illumination in the scene.

⟨*Allow integrators to do preprocessing for the scene*⟩ ≡ 27
```
surfaceIntegrator->Preprocess(scene, camera, this);
volumeIntegrator->Preprocess(scene, camera, this);
```

Before rendering starts, the `Render()` method also creates a `Sample` object, into which the `Sampler` will store the samples it generates during the main loop. Because the number and types of samples that need to be generated are partially dependent on the integrators, the `Sample` constructor takes pointers to the integrators so they can be queried for their requirements. See Section 7.2.1 for more information about how integrators request particular sets of samples.

⟨*Allocate and initialize* `sample`⟩ ≡ 27
```
Sample *sample = new Sample(sampler, surfaceIntegrator,
                            volumeIntegrator, scene);
```

Camera 302
Film 403
Integrator 740
Integrator::Preprocess() 740
PhotonIntegrator 802
Sample 343
Sampler 340
SamplerRenderer 25
SamplerRenderer::Render() 27
Scene 22

Now the main rendering loop begins. So that rendering can proceed in parallel on systems with multiple processing cores, the process of generating the image is decomposed into a set of SamplerRendererTasks where each SamplerRendererTask is responsible for computing the samples in a small rectangular subset of the image. The SamplerRendererTask inherits from the Task abstract base class defined in Appendix A.9.4.

Tasks represent independent work items that are all immediately ready for execution; they can be run in any order, on any available processor. This decomposition is the foundation of how pbrt renders images in parallel. Section 1.3.5 discusses implications to the system's design due to this decomposition, and Appendix A.9 goes into more depth on performance issues related to parallelism and presents the classes and utility methods pbrt uses in its parallel implementation.

The EnqueueTasks() function takes an array of tasks and runs them on all of the processors in the system. It returns immediately to the calling thread, allowing the tasks to execute asynchronously; here there is no other work to do while the image is rendered, so the Render() method here immediately calls WaitForAllTasks(), which suspends the calling thread of execution until all of the enqueued tasks have finished.

⟨*Create and launch* SamplerRendererTasks *for rendering image*⟩ ≡ 27
```
⟨Compute number of SamplerRendererTasks to create for rendering 29⟩
vector<Task *> renderTasks;
for (int i = 0; i < nTasks; ++i)
    renderTasks.push_back(new SamplerRendererTask(scene, this, camera,
                                sampler, sample, nTasks-1-i, nTasks));
EnqueueTasks(renderTasks);
WaitForAllTasks();
for (uint32_t i = 0; i < renderTasks.size(); ++i)
    delete renderTasks[i];
```

There are two factors to trade off in deciding how many SamplerRendererTasks to create: load-balancing and per-task overhead. On one hand, we'd like to have significantly more tasks than there are processors in the system: consider a four-core computer running only four SamplerRendererTasks. In general, it's likely that some of the SamplerRendererTasks will take less processing time than others; the ones that are responsible for parts of the image where the scene is relatively simple will usually take less processing time than parts of the image where the scene is relatively complex. Therefore, if the number of tasks was equal to the number of processors, some processors would finish before others and sit idle while waiting for the processor that had the longest running task. Figure 1.18 illustrates this idea; it shows the distribution of execution time for the SamplerRendererTasks created to render the shiny sphere scene in Figure 1.7. The longest-running task took 151 times longer than the slowest.

On the other hand, having too many tasks is also inefficient. There is a (small) fixed overhead for launching each task, and the more tasks there are, the more times this overhead must be paid. The computation of nTasks here attempts to trade off these two factors. It ensures that there are at least 32 tasks for each processing core in the system while also computing how many tasks would be needed so that each one would work

EnqueueTasks() 1041
SamplerRendererTask 29
Task 1041
WaitForAllTasks() 1041

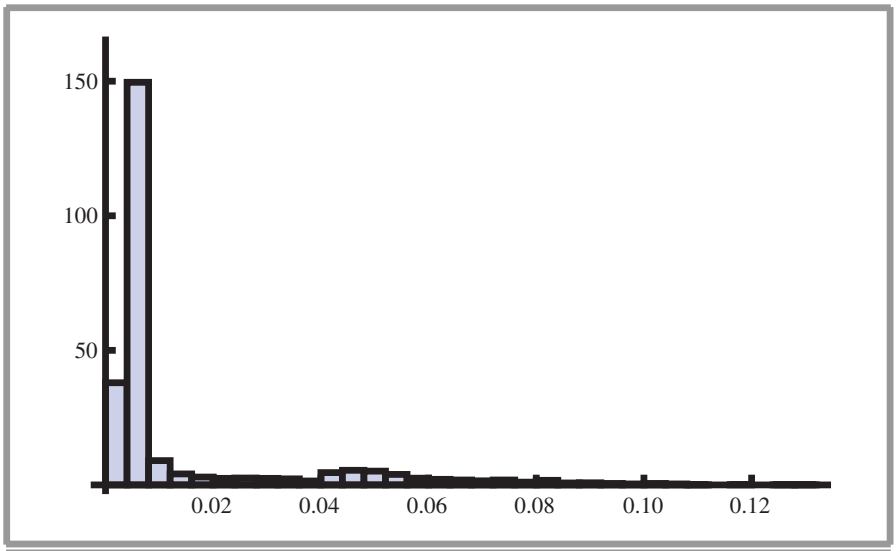

Figure 1.18: Histogram of Time Spent Running Each `SamplerRendererTask` for the Scene in Figure 1.7. The horizontal axis measures time in seconds. Note the wide variation in execution time for task execution, illustrating that different parts of the image required substantially different amounts of computation.

on an approximately 16 × 16-pixel block of the image. Whichever value gives a larger number of tasks is used. Finally, this value is rounded up to be a power of two, which will make it easier for the image decomposition code in the following to break the image into rectangular regions.

⟨*Compute number of* `SamplerRendererTask`*s to create for rendering*⟩ ≡ 28
```
int nPixels = camera->film->xResolution * camera->film->yResolution;
int nTasks = max(32 * NumSystemCores(), nPixels / (16*16));
nTasks = RoundUpPow2(nTasks);
```

After all of the rendering tasks have finished, the `SamplerRenderer`'s `Render()` method deletes the `Sample` and calls the `Film`'s method to write the image out to disk.

⟨*Clean up after rendering and store final image*⟩ ≡ 27
```
delete sample;
camera->film->WriteImage();
```

Now we will discuss the implementation of `SamplerRendererTask`.

⟨*SamplerRendererTask Declarations*⟩ ≡
```
class SamplerRendererTask : public Task {
public:
    ⟨SamplerRendererTask Public Methods 30⟩
private:
    ⟨SamplerRendererTask Private Data 30⟩
};
```

Camera::film 302
Film 403
Film::WriteImage() 404
Film::xResolution 403
Film::yResolution 403
NumSystemCores() 1041
RoundUpPow2() 1002
Sample 343
SamplerRenderer 25
SamplerRendererTask 29
Task 1041

Because the SamplerRendererTask runs independently of the SamplerRenderer::Render() method, all of the variables and data that it needs must be passed to its constructor. These values are all stored in SamplerRendererTask member variables for access when the task actually runs. This is a typical idiom for tasks in the system; we will usually elide this part of Task implementations for brevity. Note that it is also given a task number, taskNum, and the total number of tasks launched, taskCount. From these two values, the task will later be able to determine which part of the image it is responsible for computing.

⟨*SamplerRendererTask Public Methods*⟩ ≡ 29
```
SamplerRendererTask(const Scene *sc, Renderer *ren, Camera *c,
                    Sampler *ms, Sample *sam, int tn, int tc)
{
    scene = sc; renderer = ren; camera = c; mainSampler = ms;
    origSample = sam; taskNum = tn; taskCount = tc;
}
```

⟨*SamplerRendererTask Private Data*⟩ ≡ 29
```
const Scene *scene;
const Renderer *renderer;
Camera *camera;
Sampler *mainSampler;
Sample *origSample;
int taskNum, taskCount;
```

When the task system in Appendix A.9.4 decides to run this task on a particular processor, the SamplerRendererTask::Run() method will be called and the task can start its work. The task does a little bit of setup work, determining which part of the film plane it is responsible for and allocating space for some temporary data before using the Sampler to generate image samples, the Camera to determine corresponding rays leaving the film plane, and the Integrators to compute radiance along those rays arriving at the film.

⟨*SamplerRendererTask Definitions*⟩ ≡
```
void SamplerRendererTask::Run() {
    ⟨Get sub-Sampler for SamplerRendererTask 31⟩
    ⟨Declare local variables used for rendering loop 31⟩
    ⟨Allocate space for samples and intersections 31⟩
    ⟨Get samples from Sampler and update image 32⟩
    ⟨Clean up after SamplerRendererTask is done with its image region⟩
}
```

As the scene description is parsed, a Sampler that generates image and integration samples for the entire image is created. Here, the SamplerRendererTask uses the Sampler::GetSubSampler() method to get a new Sampler that only generates samples for the subset of the image that the SamplerRenderer is responsible for. The GetSubSampler() method uses the task number and the total number of tasks passed to the SamplerRendererTask constructor to determine which part of the image the returned subsampler should generate samples for. This method may return NULL, because work is decomposed by image pixels—for example, given a 1×1-pixel image, only one task will have any work to do. Any additional tasks that were created will just return immediately. (It would perhaps be

Camera 302
Integrator 740
Renderer 24
Sample 343
Sampler 340
SamplerRenderer::Render() 27
SamplerRendererTask 29
SamplerRendererTask::Run() 30
Scene 22
Task 1041

better to not create the extra tasks at all in this case, though this is an infrequent enough situation that it's just as well to catch it here.)

⟨*Get sub*-`Sampler` *for* `SamplerRendererTask`⟩ ≡ 30
```
Sampler *sampler = mainSampler->GetSubSampler(taskNum, taskCount);
if (!sampler)
    return;
```

During rendering, the `Run()` method needs two additional task-local variables: a `MemoryArena` and a `RNG`. The `MemoryArena` is passed to `Integrators` for allocating small amounts of temporary memory during rendering. Rather than call the system's regular memory allocation routines (e.g. `malloc()` or `new`) to do this, the system instead uses the `MemoryArena` class to manage pools of memory for this purpose. Using the `MemoryArena` for this purpose both enables higher-performance allocation than is provided by the standard library routines and also simplifies freeing allocated data. The `RNG` is made available to `Integrators` for generating pseudo-random numbers for Monte Carlo sampling.

Neither of these classes is safe to be used by multiple threads concurrently without additional synchronization; for efficiency, it's better to have unique instances of them for each task so that they can just be used directly.

⟨*Declare local variables used for rendering loop*⟩ ≡ 30
```
MemoryArena arena;
RNG rng(taskNum);
```

The `Sampler` may provide one or more sample values when asked to generate samples. For some sample-generation algorithms, it's easier to generate a set of sample values all at once than to generate them one at a time. The `Sampler::MaximumSampleCount()` method returns an upper bound on the number of samples it will return at once. Given this bound, that number of `Samples` are created to store the values returned by the `Sampler`. For each sample, it is necessary to store the corresponding ray (represented here by the `RayDifferential` class), spectra for the radiance along the ray `Ls`, the transmittance along the ray `Ts`, and an `Intersection` object for each ray intersection. All of these arrays are deallocated at the end of the `Render()` method.

⟨*Allocate space for samples and intersections*⟩ ≡ 30
```
int maxSamples = sampler->MaximumSampleCount();
Sample *samples = origSample->Duplicate(maxSamples);
RayDifferential *rays = new RayDifferential[maxSamples];
Spectrum *Ls = new Spectrum[maxSamples];
Spectrum *Ts = new Spectrum[maxSamples];
Intersection *isects = new Intersection[maxSamples];
```

Each time through the loop, `Sampler::GetMoreSamples()` is called to initialize the `samples` array with one or more image sample values; this method returns the number of samples it initialized, or zero when it has finished generating all of the samples for the region of the image that it is responsible for. The fragments in the loop body find the corresponding camera ray for each sample and pass it to the integrators to compute the radiance along the ray arriving at the film plane. Finally, they add the samples' contributions to the final image and free any temporary memory allocated in the `MemoryArena` during the course

Integrator 740
Intersection 186
MemoryArena 1015
RayDifferential 69
RNG 1003
Sample 343
Sample::Duplicate() 346
Sampler 340
Sampler::
GetMoreSamples() 340
Sampler::GetSubSampler() 341
Sampler::
MaximumSampleCount() 341
SamplerRendererTask::
mainSampler 30
SamplerRendererTask::
origSample 30
SamplerRendererTask::
taskCount 30
SamplerRendererTask::
taskNum 30
Spectrum 263

of computing results for the rays. (The MemoryArena here therefore shouldn't be used to allocate any memory with a longer lifetime than needed to compute the result for a single ray.)

⟨*Get samples from* Sampler *and update image*⟩ ≡ 30
```
int sampleCount;
while ((sampleCount = sampler->GetMoreSamples(samples, rng)) > 0) {
    ⟨Generate camera rays and compute radiance along rays 32⟩
    ⟨Report sample results to Sampler, add contributions to image 33⟩
    ⟨Free MemoryArena memory from computing image sample values 33⟩
}
```

⟨*Generate camera rays and compute radiance along rays*⟩ ≡ 32
```
for (int i = 0; i < sampleCount; ++i) {
    ⟨Find camera ray for sample[i] 32⟩
    ⟨Evaluate radiance along camera ray 33⟩
}
```

The Camera interface provides two main methods: Camera::GenerateRay(), which returns the ray for a given image sample position, and Camera::GenerateRayDifferential(), which returns a *ray differential*, which incorporates information about the rays that the Camera would generate for samples that are one pixel away on the image plane in both the x and y directions. Ray differentials are used to get better results from some of the texture functions defined in Chapter 10, making it possible to compute how quickly a texture varies with respect to the pixel spacing, a key component of texture antialiasing. While the Ray class holds just the origin and direction of a single ray, RayDifferential inherits from Ray so that it has not only those member variables but also two additional Rays, rx and ry, to hold these neighbors. One important detail is that the direction vector of the generated ray must be of unit length; most of the integrators depend on this property.

The GenerateRayDifferential() method is passed a Sample and a pointer to a ray differential; it initializes all of the fields of the RayDifferential based on the contents of the sample to give the differential rays as if a single sample is being taken for each pixel (i.e., with an implicit assumption that image samples are spaced one pixel apart). However, when rendering high-quality images, many samples are often averaged together to compute each pixel value. Therefore, the ScaleDifferentials() methods scales the differential rays to account for the actual spacing between samples on the film plane.

The camera also returns a floating-point weight associated with the ray. For simple camera models, each ray is weighted equally, but more complex Cameras that more accurately model the process of image formation by lens systems may generate some rays that contribute more than others. Such a camera model might simulate the effect of less light arriving at the edges of the film plane than at the center, an effect called *vignetting*. The returned weight will be used here to scale the ray's contribution to the image.

⟨*Find camera ray for* sample[i]⟩ ≡ 32
```
float rayWeight = camera->GenerateRayDifferential(samples[i], &rays[i]);
rays[i].ScaleDifferentials(1.f / sqrtf(sampler->samplesPerPixel));
```

Camera 302
Camera::GenerateRay() 303
Camera::GenerateRayDifferential() 303
MemoryArena 1015
Ray 66
RayDifferential 69
RayDifferential::ScaleDifferentials() 70
Sampler 340
Sampler::GetMoreSamples() 340
Sampler::samplesPerPixel 340
SamplerRendererTask::camera 30

Now that we have a ray, the next task is to determine the amount of light arriving at the image plane along that ray (its *radiance*). The usual notation for radiance arriving along a ray is L_i, so the method to compute radiance is `Renderer::Li()`. Radiance values are represented here with the `Spectrum` class, which is `pbrt`'s abstraction for energy distributions that vary over wavelength—in other words, *color*.

⟨*Evaluate radiance along camera ray*⟩ ≡ 32
```
if (rayWeight > 0.f)
    Ls[i] = rayWeight * renderer->Li(scene, rays[i], &samples[i], rng,
                                     arena, &isects[i], &Ts[i]);
else {
    Ls[i] = 0.f;
    Ts[i] = 1.f;
}
```
⟨*Issue warning if unexpected radiance value returned*⟩

A common side effect of bugs in the rendering process is that impossible radiance values are computed. For example, division by zero results in radiance values equal either to the IEEE floating-point infinity or "not a number" value. The renderer looks for this possibility, as well as for spectra with negative contributions, and prints an error message when it encounters them. Here we won't include the fragment that does this, ⟨*Issue warning if unexpected radiance value returned*⟩. See the implementation in `renderers/samplerrenderer.cpp` if you're interested in its details.

After the radiance carried by the rays is known, the image can be updated. Before this happens, the `Sampler::ReportResults()` method is used to pass the radiance values and information about the intersections found back to the `Sampler`; this gives the sampler a chance to incorporate information from the results of these samples into the samples it generates later. (For example, it could generate extra samples in pixels that have a lot of detail.) This method returns `true` if this group of samples should be added to the image, or `false` if it should be discarded. For some adaptive sampling algorithms, the sampler may want to discard the initial set of samples and generate new ones in their stead.

If the samples are to be added to the image, the `Film::AddSample()` method updates the pixels in the image given the results from a sample. The details of this process are explained in Sections 7.7 and 7.8.

Camera::film 302
Film::AddSample() 403
MemoryArena 1015
MemoryArena::FreeAll() 1017
Renderer::Li() 25
Sampler 340
Sampler::ReportResults() 341
SamplerRendererTask::camera 30
SamplerRendererTask::renderer 30
SamplerRendererTask::scene 30
Spectrum 263

⟨*Report sample results to* `Sampler`*, add contributions to image*⟩ ≡ 32
```
if (sampler->ReportResults(samples, rays, Ls, isects, sampleCount))
    for (int i = 0; i < sampleCount; ++i)
        camera->film->AddSample(samples[i], Ls[i]);
```

After processing a group of samples, all of the allocated memory in the `MemoryArena` is freed together when `MemoryArena::FreeAll()` is called. Section 9.1.1 discusses one use of the `MemoryArena` during rendering, and Section A.5.4 has the `MemoryArena` implementation.

⟨*Free* `MemoryArena` *memory from computing image sample values*⟩ ≡ 32
```
arena.FreeAll();
```

We won't include the final code fragment from SamplerRendererTask::Run(), ⟨*Clean up after* SamplerRendererTask *is done with its image region*⟩ here, as it just frees the memory allocated for samples, rays, and intersections.

Recall that the SamplerRenderer::Li() method is called by the SamplerRendererTask::Run() method to compute the radiance along the given ray. It is given a Sample to pass along to the integrators for their use in their computations and a MemoryArena for their temporary memory needs. Information about the intersection point and the transmittance due to volume scattering along the ray may be returned to the caller, if non-NULL values are passed for the corresponding parameters.

The Scene::Intersect() method finds the first surface that the ray intersects by passing the request on to an accelerator Primitive. The accelerator performs ray-primitive intersection tests with the geometric Primitives that the ray potentially intersects, using each shape's Shape::Intersect() routine. If an intersection is found, this method returns true and returns a filled-in Intersection object. The SamplerRenderer::Li() method then calls SurfaceIntegrator::Li() to compute the outgoing radiance L_o from the first surface that the ray intersects and then stores the result in Li. It then invokes VolumeIntegrator::Transmittance() to compute the fraction of light T that is extinguished between the surface and the camera due to participating media. Finally, VolumeIntegrator::Li() determines the radiance L_v added along the ray due to interactions with participating media. The net effect of these interactions is $TL_i + L_v$; this calculation is explained further in Section 16.1.

⟨*SamplerRenderer Method Definitions*⟩ +≡

```
Spectrum SamplerRenderer::Li(const Scene *scene,
        const RayDifferential &ray, const Sample *sample, RNG &rng,
        MemoryArena &arena, Intersection *isect, Spectrum *T) const {
    ⟨Allocate local variables for isect and T if needed 35⟩
    Spectrum Li = 0.f;
    if (scene->Intersect(ray, isect))
        Li = surfaceIntegrator->Li(scene, this, ray, *isect, sample,
                                   rng, arena);
    else {
        ⟨Handle ray that doesn't intersect any geometry 35⟩
    }
    Spectrum Lvi = volumeIntegrator->Li(scene, this, ray, sample, rng,
                                        T, arena);
    return *T * Li + Lvi;
}
```

If the caller does pass NULL values for isect and T parameters, then scratch instances of these objects are allocated on the stack in this method so that the integrators can just assume that the corresponding values are non-NULL.

Intersection 186
MemoryArena 1015
Primitive 185
RayDifferential 69
RNG 1003
Sample 343
SamplerRenderer 25
SamplerRenderer::Li() 34
SamplerRenderer::surfaceIntegrator 26
SamplerRenderer::volumeIntegrator 26
SamplerRendererTask 29
SamplerRendererTask::Run() 30
Scene 22
Scene::Intersect() 23
Shape::Intersect() 111
Spectrum 263
SurfaceIntegrator::Li() 741
VolumeIntegrator::Li() 876
VolumeIntegrator::Transmittance() 876

⟨*Allocate local variables for* `isect` *and* `T` *if needed*⟩ ≡ 34
```
Spectrum localT;
if (!T) T = &localT;
Intersection localIsect;
if (!isect) isect = &localIsect;
```

Rays that don't hit any geometry may still carry radiance—certain types of light sources may not be associated with any geometry but can still contribute radiance to rays that do not hit anything. For example, the Earth's sky illuminates points on the Earth's surface with blue light, even though there isn't geometry associated with the sky per se. Therefore, for rays that do not hit anything, each light's `Light::Le()` method is called so that these particular lights can contribute to the ray's radiance. Most light sources will not contribute in this way, but in certain cases this is very useful. See Section 12.5 for an example of such a light.

⟨*Handle ray that doesn't intersect any geometry*⟩ ≡ 34
```
for (uint32_t i = 0; i < scene->lights.size(); ++i)
   Li += scene->lights[i]->Le(ray);
```

It is also useful to be able to independently compute the attenuation along a ray due to participating media; the `SamplerRenderer::Transmittance()` method computes this quantity by forwarding the request to the `VolumeIntegrator::Transmittance()` method.

⟨*SamplerRenderer Method Definitions*⟩ +≡
```
Spectrum SamplerRenderer::Transmittance(const Scene *scene,
        const RayDifferential &ray, const Sample *sample, RNG &rng,
        MemoryArena &arena) const {
    return volumeIntegrator->Transmittance(scene, this, ray, sample,
                                           rng, arena);
}
```

1.3.5 PARALLELIZATION OF `pbrt`

As of the writing of the second edition of this book, it's difficult to buy a new laptop or desktop computer with only one processing core. The computer systems of today and of the future will increasingly feature multiple processing cores, both on CPUs and on highly parallel throughput processors like GPUs. This development in computer architecture places a new burden on software developers, as significant increases in performance over time are now only available to programs that can run in parallel, with many separate threads of execution performing computation simultaneously.

This burden is a substantial one; parallel programming is notoriously difficult. When two computations that the programmer believes are independent are executing simultaneously but then interact unexpectedly, the program may crash or generate unexpected results. However, the bug may not manifest itself again if the program is run again, perhaps due to those particular computations not running simultaneously during the next

Intersection 186
Light::Le() 631
MemoryArena 1015
RayDifferential 69
RNG 1003
Sample 343
SamplerRenderer 25
SamplerRenderer::
Transmittance() 35
SamplerRenderer::
volumeIntegrator 26
Scene 22
Scene::lights 23
Spectrum 263
VolumeIntegrator::
Transmittance() 876

run. Fortunately, increasingly good tools to help developers find these sorts of interactions are being developed.[4]

For a parallel program to scale well to increasingly large numbers of processors, it needs to be able to provide a substantial amount of independent computation to the system. However, any computation dependent on results of prior computation can't be run concurrently with the computation it depends on. Some researchers have claimed that ray tracing is in the class of *embarrassingly parallel* applications. This observation is partially correct, in that there is in fact an enormous amount of independent computation available in most ray-tracing algorithms; each pixel could potentially be rendered by a separate processor, offering scaling up to computers with millions of processors. However, as we'll see at a number of places in the implementation of `pbrt`, the parallelization of a reasonably complete software system is more difficult than the parallelization of a small proof-of-concept research system.

We assume that the system has some number of processing *cores*, each of which can run a separate thread of execution. A *task system*, implemented in Appendix A.9.4, launches a hardware thread on each core in the system. These threads poll a *task queue* for work. As soon as code in `pbrt` adds tasks to the task queue, processing cores can start to remove tasks from the queue and execute them.

In `pbrt` we assume that the computation is running on processors that provide *coherent shared memory*. (This is true of today's CPUs and is likely to continue to be provided by future CPUs.) The main idea is that all of the threads can read and write to a common set of memory locations and that changes to memory made by one thread will be seen by other threads. This property greatly simplifies the implementation of the system as there's no need to explicitly communicate data between tasks.

Data Races and Coordination

Although coherent shared memory relieves tasks of the need to explicitly communicate data with each other, they still need to *coordinate* their access to shared data; a danger of coherent shared memory is *data races*. If two threads of execution modify the same location in memory without coordination between the two of them, the program will almost certainly compute incorrect results or even crash. Consider the example of two processors simultaneously running the following innocuous-looking code, where `globalCounter` starts with a value of two:

```
extern int globalCounter;
if (--globalCounter == 0) {
    printf("done!\n");
}
```

Because the two threads don't coordinate their reading and writing of `globalCounter`, it is possible that "done" will be printed zero, one, or even two times! The assembly instructions generated by the compiler are likely to correspond to steps something like the following:

4 We found the open source tool `helgrind`, part of the `valgrind` suite of tools, instrumental for helping to find bugs in the `pbrt` implementation as we were developing it.

```
extern int globalCounter;
int temp = globalCounter;
temp = temp - 1;
globalCounter = temp;
if (globalCounter == 0) {
    printf("done!\n");
}
```

Now, consider different ways this code could be executed on two processors. For example, the second processor could start executing slightly after the first one, but the first one could go idle for a few cycles after executing the first few instructions:

```
Thread A                        Thread B
int temp = globalCounter;
temp = temp - 1;                (idle)
globalCounter = temp;
                                int temp = globalCounter;
(idle)                          temp = temp - 1;
                                globalCounter = temp;
if (globalCounter == 0) {       if (globalCounter == 0) {
    printf("done!\n");             printf("done!\n");
}                               }
```

Many unpredictable events can cause these sorts of execution bubbles, ranging from the OS interrupting the thread to cache misses.

In this ordering, both threads read the value of zero from `globalCounter` and both execute the `printf()` call. In this case, the error is not fatal, but if instead the system was freeing resources in the `if` block, then it would attempt to free the same resources twice, which would very likely cause a crash. Consider now this potential execution order:

```
Thread A                        Thread B
int temp = globalCounter;       int temp = globalCounter;
temp = temp - 1;                temp = temp - 1;
globalCounter = temp;           (idle)
(idle)                          globalCounter = temp;
if (globalCounter == 0) {       if (globalCounter == 0) {
    printf("done!\n");             printf("done!\n");
}                               }
```

In this case, `globalCounter` ends up with a value of one, and neither thread executes the `if` block. These examples illustrate the principle that when multiple threads of execution are accessing shared modified data, they must somehow synchronize their access. Two main mechanisms are available for doing this type of synchronization: mutual exclusion and atomic operations.

Mutex 1038

MutexLock 1039

Mutual exclusion is implemented with `Mutex` objects in `pbrt`. A `Mutex` can be used to protect access to a memory location, ensuring that only one thread can update it at a time. Consider the following updated version of the previous computation; here a `MutexLock`

object acquires a lock on the mutex and releases it when it goes out of scope at the final brace.

```
extern int globalCounter;
extern Mutex globalCounterMutex;
{ MutexLock lock(globalCounterMutex);
  int temp = globalCounter;
  temp = temp - 1;
  globalCounter = temp;
  if (globalCounter == 0) {
      printf("done!\n");
  }
}
```

If two threads are executing this code and try to acquire the mutex at the same time, then the mutex will allow only one of them to proceed, stalling the other one in the `MutexLock` constructor. Only when the first thread has finished the computation and its `MutexLock` goes out of scope, releasing the lock on the mutex, is the second thread able to acquire the mutex itself and continue the computation.

```
Thread A                                  Thread B
{ MutexLock lock(globalCounterMutex);    { MutexLock lock(globalCounterMutex);
  int temp = globalCounter;                  (stalled by mutex)
  .
  .
  .
} (mutex released)
                                             (mutex acquired)
                                             int temp = globalCounter;
                                             .
                                             .
                                             .
                                           } (mutex released)
```

With correct mutual exclusion here, the `printf()` will only be executed once, no matter what the ordering of execution between the two threads is.

Atomic memory operations (or *atomics*) are the other option for correctly performing this type of memory update with multiple threads. Atomics are a feature provided by almost all CPU architectures; they are machine instructions that guarantee that their respective memory updates will be performed in a single transaction. ("Atomic" in this case refers to the notion that the memory updates are indivisible.) The implementations of atomic operations in `pbrt` are defined in Appendix A.9.2. Using atomics, the computation above could be written to use the `AtomicAdd()` increment operation (incrementing by −1 in this case) as below:

```
extern AtomicInt32 globalCounter;
if (AtomicAdd(&globalCounter, -1) == 0) {
  printf("done!\n");
}
```

AtomicAdd() 1038
MutexLock 1039

`AtomicAdd()` adds the given value (here, -1) to the given variable (`globalCounter`) and returns the new value of the variable. Using an atomic operation ensures that if two threads simultaneously try to update the variable then not only will the final value of the variable be the expected value but each thread will be returned the value of the variable after its update alone. In this example, then, `globalCounter` will end up with a value of zero, as expected, with one thread guaranteed to have the value one returned from the `AtomicAdd()` call and the other thread guaranteed to have zero returned.

When there is a choice between atomic operations and mutexes, atomics are generally preferable, in that they are generally more efficient. However, if more than a single memory location needs to be updated then a mutex must generally be used.[5] Note that both of these approaches—atomics and mutexes—are conventions that the programmer must ensure are consistently adhered to. If just a single code-path updates a shared variable without consistently holding a mutex or using an atomic (whichever approach was chosen for updates to that location), then the program may fail under some circumstances.

Section A.9 in Appendix A has more information about parallel programming, with additional details on performance issues and pitfalls, as well as the various utility classes and routines used in the parallel implementation of `pbrt`.

Conventions in pbrt

In Section 1.3.4, we decomposed the process of rendering an image into a number of tasks, where each task was repsonsible for a small rectangular portion of the image (this approach is sometimes called *screen-space decomposition*). When rendering begins, each processor takes a task from the task queue and does the work associated with it, until no more tasks remain. These tasks are, by design, independent, so they can be executed in any order on however many processors are available in the system. However, care is necessary to ensure that there are no synchronization errors in the data accesses made by the tasks.

In `pbrt` (as is the case for most ray tracers) the vast majority of data at render-time is read only (for example, the scene description and texture maps). All of the parsing of the scene file and creation of the scene representation in memory is done with a single thread of execution, so there are no synchronization issues during that phase of execution. During rendering, concurrent read access to all of the read-only data by multiple threads works directly; we only need to be concerned with situations where data in memory is being updated.

Updating the output image is an important instance of a case where multiple tasks need to update shared data. Although the rendering work is distributed to tasks by image decomposition, under some circumstances tasks will need to update pixel data that is nominally being computed by other tasks—see Section 7.7 for the filtering theory behind this issue and Section 7.8.2 for the implementation of the image update code that deals with synchronization between threads. There, if we can be sure that only a single task will be updating the image data for a given pixel then the code just updates it directly, but if multiple tasks need to update it then atomic operations are used instead.

AtomicAdd() 1038

5 A generalization of atomic operations, *transactional memory*, makes possible more extensive sequences of memory updates, though it isn't yet available on mainstream processors.

Another important case of multiple tasks updating shared data is preprocessing work by some Integrators that build data structures for use during rendering before rendering begins. Some of them do substantial amounts of work during this phase, and in turn they break their preprocessing down into many tasks. See the implementations of the Preprocess() methods in Sections 15.5 (Irradiance Caching) and 15.6 (Photon Mapping) for examples.

The final instance of the shared data update issue is in data structures that are built on demand during rendering. For example, the GridAccel defined in Section 4.3 lazily builds a data structure for accelerating ray-primitive intersection computations. Because this data structure is shared by all of the currently executing tasks, updates to it during rendering need to be coordinated. (See in particular the use of a mutex in the grid traversal code in Section 4.3.2.)

When adding new code to pbrt, it's important to make sure to not inadvertently add code that modifies shared data without proper synchronization. In many cases, this can easily happen inadvertently—for example, when adding a new Shape implementation, the Shape will normally only perform read accesses to its member variables after it has been created. Sometimes, however, shared data may be inadvertently introduced. Consider the following code idiom, often seen in single-threaded code:

```
static bool firstCall = true;
if (firstCall) {
    .
    .  additional initialization
    .
    firstCall = false;
}
```

This code is unsafe with multiple threads of execution, as multiple threads may see the value of firstCall as true and all execute the initialization code. Writing this safely requires either atomic operations or mutexes. (This particular idiom probably needs a mutex, as it is likely that we would want other threads of execution that reach this test to stall until the additional intialization had been completed.)

Reentrancy Expectations in pbrt

Many methods of classes in pbrt are required to be *reentrant*—safe for multiple threads of execution to call concurrently. Particular instances of these methods must either be safe naturally due to not updating shared global data or using mutexes or atomic operations to safely perform any updates that are needed.

As a general rule, the low-level classes and structures in the system are not reentrant. For example, the Point class, which stores three float values to represent a point in 3D space, is not safe for multiple threads to call methods that modify it at the same time. (Multiple threads can use Points as read-only data simultaneously, of course.) The runtime overhead to make Point thread-safe would have a substantial effect on performance with little benefit in return.

The same is true for classes like Vector, Normal, Spectrum, Transform, Quaternion, and DifferentialGeometry. These classes are usually either created at scene construction time

DifferentialGeometry 102
GridAccel 196
Integrator 740
Normal 65
Point 63
Quaternion 92
Shape 108
Spectrum 263
Transform 76
Vector 57

and then used as read-only data or allocated on the stack during rendering and used only by a single thread.

The utility classes `MemoryArena` (used for high-performance temporary memory allocation) and `RNG` (pseudo-random number generation) are also not safe for use by multiple threads; these classes store state that is modified by their method calls, and the overhead from protecting modification to their state with mutual exclusion would be excessive relative to the amount of computation they perform. Consequently, in code like the `SamplerRendererTask::Run()` method above, the implementation allocates these as private per-thread instances of these classes allocated on the stack.

With two exceptions, implementations of the higher-level abstract base classes listed in Table 1.1 are all expected to be safe for multiple threads to use simultaneously. With a little care, it is usually straightforward to implement specific instances of these base classes so they don't modify any shared state in their methods.

The first exception is the `Integrator Preprocess()` and `RequestSamples()` methods. These are called by the system during scene construction, and implementations of them generally modify shared state in the `Integrator` implementation—for example, by building data structures that represent the distribution of illumination in the scene. Therefore, it's helpful to allow the implementor to assume that only a single thread will call into these methods. (This is a separate issue from the consideration that for implementations of these methods that are computationally intensive the implementation of them will often launch multiple `Task`s to do the computation in parallel.)

The second exception is the `Sampler::GetMoreSamples()` and `Sampler::ReportResults()` methods—these are also not expected to be thread safe. This is another instance where this requirement would impose an excessive performance and scalability impact; many threads simultaneously trying to get samples from a single `Sampler` would limit the system's overall performance. Therefore, as described in Section 1.3.4, a unique `Sampler` is created for each rendering task with `Sampler::GetSubSampler()`; this sampler can then be used by just the single task, without any mutual exclusion overhead.

All stand-alone functions in `pbrt` are reentrant (as long as multiple threads don't pass pointers to the same data to them).

1.3.6 AN INTEGRATOR FOR WHITTED RAY TRACING

Chapters 15 and 16 include the implementations of many different surface and volume integrators, based on a variety of algorithms with differing levels of accuracy. Here we will present a surface integrator based on Whitted's ray-tracing algorithm. This integrator accurately computes reflected and transmitted light from specular surfaces like glass, mirrors, and water, although it doesn't account for other types of indirect lighting effects like light bouncing off a wall and illuminating a room. The more complex integrators later in the book build on the ideas in this integrator to implement more sophisticated light transport algorithms. The `WhittedIntegrator` class can be found in the `integrators/whitted.h` and `integrators/whitted.cpp` files in the `pbrt` distribution.

Integrator 740
MemoryArena 1015
RNG 1003
Sampler 340
Sampler::
GetMoreSamples() 340
Sampler::GetSubSampler() 341
Sampler::ReportResults() 341
SamplerRendererTask::Run() 30
Task 1041
WhittedIntegrator 42

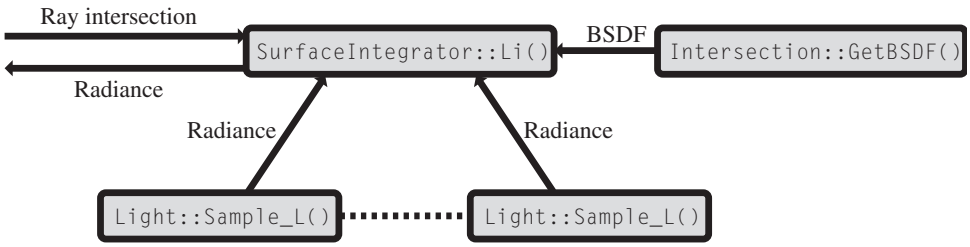

Figure 1.19: Class Relationships for Surface Integration. The main rendering loop passes a camera ray and information about its intersection point to the `SurfaceIntegrator`, which returns the radiance along that ray arriving at the film plane. The integrator finds the material properties at the intersection point in the form of a `BSDF` and uses the `Lights` in the `Scene` to determine the illumination there. Together, these give the information needed to compute the radiance reflected back along the ray at the intersection point.

⟨*WhittedIntegrator Declarations*⟩ ≡
```
class WhittedIntegrator : public SurfaceIntegrator {
public:
    ⟨WhittedIntegrator Public Methods⟩
private:
    ⟨WhittedIntegrator Private Data 42⟩
};
```

The key method that all surface integrators must provide is `SurfaceIntegrator::Li()`, which returns the radiance along a ray. Figure 1.19 summarizes the data flow among the main classes used during integration at surfaces.

⟨*WhittedIntegrator Method Definitions*⟩ ≡
```
Spectrum WhittedIntegrator::Li(const Scene *scene,
        const Renderer *renderer, const RayDifferential &ray,
        const Intersection &isect, const Sample *sample, RNG &rng,
        MemoryArena &arena) const {
    Spectrum L(0.);
    ⟨Compute emitted and reflected light at ray intersection point 43⟩
    return L;
}
```

The Whitted integrator works by recursively evaluating radiance along reflected and refracted ray directions. It stops the recursion at a predetermined maximum depth, `WhittedIntegrator::maxDepth`. By default, the maximum recursion depth is five. Without this termination criterion, the recursion might never terminate (imagine, for example, a hall-of-mirrors scene). This member variable is initialized based on parameters set in the scene description file in the `WhittedIntegrator` constructor, which we will not show here.

⟨*WhittedIntegrator Private Data*⟩ ≡ 42
```
int maxDepth;
```

BSDF 478
Intersection 186
Light 606
MemoryArena 1015
RayDifferential 69
Renderer 24
RNG 1003
Sample 343
Scene 22
Spectrum 263
SurfaceIntegrator 740
SurfaceIntegrator::Li() 741
WhittedIntegrator 42
WhittedIntegrator::
  maxDepth 42

⟨*Compute emitted and reflected light at ray intersection point*⟩ ≡ 42
```
⟨Evaluate BSDF at hit point 43⟩
⟨Initialize common variables for Whitted integrator 43⟩
⟨Compute emitted light if ray hit an area light source 43⟩
⟨Add contribution of each light source 45⟩
if (ray.depth + 1 < maxDepth) {
    ⟨Trace rays for specular reflection and refraction 46⟩
}
```

Recall that bidirectional scattering distribution functions describe how a surface reflects light arriving at its boundary; they essentially encapsulate the difference between the appearance of a mirror versus colored paint, and so forth. They are represented in pbrt by the BSDF class. pbrt provides BSDF implementations for several standard scattering functions used in computer graphics, such as Lambertian reflection and the Torrance–Sparrow microfacet model. These and other reflection models are described in Chapter 8.

The BSDF interface makes it possible to compute reflected light at a single surface point, but BSDFs may vary across a surface. Surfaces with complex material properties, such as wood or marble, have a different BSDF at each point. Even if wood is modeled as being perfectly diffuse, the color at each point will depend on the wood's grain. These spatial variations of shading parameters are described with Textures, which in turn may be described procedurally or stored in image maps (Chapter 10).

To obtain the BSDF at the hit point, the integrator calls the Intersection::GetBSDF() method:

⟨*Evaluate BSDF at hit point*⟩ ≡ 43
```
BSDF *bsdf = isect.GetBSDF(ray, arena);
```

Figure 1.20 shows a few quantities that will be used frequently in the fragments to come; p represents the position of the ray-primitive intersection, and n is the surface normal at the intersection point. The normalized direction from the hit point back to the ray origin is stored in wo; because Cameras are responsible for normalizing the direction component of generated rays, there's no need to renormalize it here. Normalized directions are denoted by the ω symbol in this book, and in pbrt's code we will use the shorthand wo for ω_o, the outgoing direction of scattered light.

BSDF 478
BSDF::dgShading 479
Camera 302
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
Intersection::GetBSDF() 484
Intersection::Le() 625
Normal 65
Point 63
Ray::d 67
Ray::depth 67
Texture 519
Vector 57
WhittedIntegrator::
  maxDepth 42

⟨*Initialize common variables for Whitted integrator*⟩ ≡ 43
```
const Point &p = bsdf->dgShading.p;
const Normal &n = bsdf->dgShading.nn;
Vector wo = -ray.d;
```

In case the ray happened to hit geometry that is emissive (such as an area light source), the integrator computes the emitted radiance by calling the Intersection::Le() method. This gives the first term of the light transport equation, Equation (1.1) on page 11. If the object is not emissive, this method returns a black spectrum.

⟨*Compute emitted light if ray hit an area light source*⟩ ≡ 43
```
L += isect.Le(wo);
```

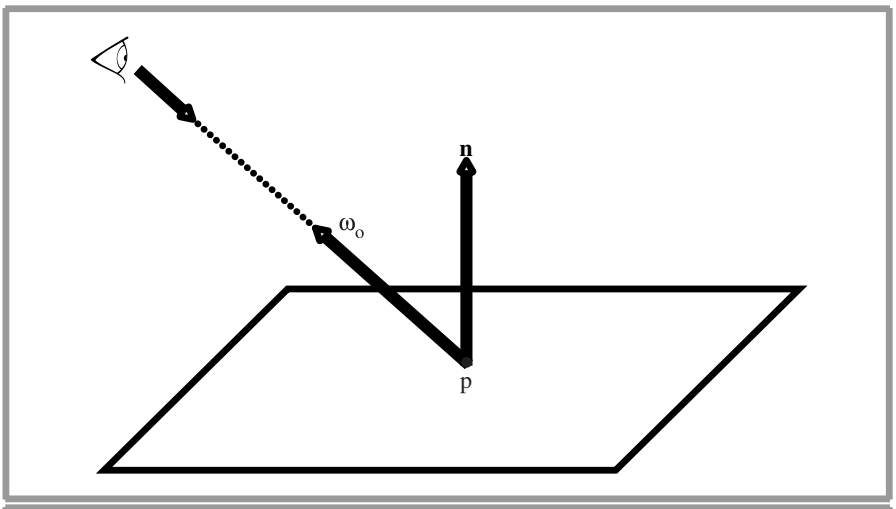

Figure 1.20: Geometric Setting for the Whitted Integrator. p is the ray intersection point and **n** is the surface normal there. The direction in which we'd like to compute reflected radiance is ω_o; it is the vector pointing in the opposite direction of the incident ray.

For each light, the integrator calls the `Light::Sample_L()` method to compute the radiance from that light falling on the surface at the point being shaded. This method also returns the direction vector from the point being shaded to the light source, which is stored in the variable `wi` (denoting an incident direction ω_i).

The spectrum returned by this method does not account for the possibility that some other shape may block light from the light and prevent it from reaching the point being shaded. Instead, it returns a `VisibilityTester` object that can be used to determine if any primitives block the surface point from the light source. This test is done by tracing a shadow ray between the point being shaded and the light to verify that the path is clear. `pbrt`'s code is organized in this way so that it can avoid tracing the shadow ray unless necessary. Specifically, this way it can first make sure that the light falling on the surface *would* be scattered in the direction ω_o if the light isn't blocked. For example, if the surface is not transmissive, then light arriving at the back side of the surface doesn't contribute to reflection.

The `Sample_L()` method also returns the probability density for the light to have sampled the direction `wi` in the `pdf` variable. This value is used for Monte Carlo integration with complex area light sources where light is arriving at the point from many directions even though just one direction is sampled here; for simple lights like point lights, the value of `pdf` is one. The details of how this probability density is computed and used in rendering are the topic of Chapters 13 to 15; in the end, the light's contribution must be divided by `pdf`, so this is done by the implementation here.

`Light::Sample_L()` 608

`VisibilityTester` 608

If the arriving radiance is nonzero and the BSDF indicates that some of the incident light from the direction ω_i is in fact scattered to the outgoing direction ω_o, then the integrator multiplies the radiance value L_i by the value of the BSDF f; the cosine term

(computed via the dot product between the ω_i vector and the surface normal); and the transmittance between the intersection point and the light. This product represents the light's contribution to the light transport equation integral, Equation (1.1), and it is added to the total reflected radiance L_o. After all lights have been considered, the integrator has computed the total contribution of *direct lighting*—light that arrives at the surface directly from emissive objects (as opposed to light that has reflected off other objects in the scene before arriving at the point).

⟨*Add contribution of each light source*⟩ ≡ 43

```
for (uint32_t i = 0; i < scene->lights.size(); ++i) {
    Vector wi;
    float pdf;
    VisibilityTester visibility;
    Spectrum Li = scene->lights[i]->Sample_L(p, isect.rayEpsilon,
        LightSample(rng), ray.time, &wi, &pdf, &visibility);
    if (Li.IsBlack() || pdf == 0.f) continue;
    Spectrum f = bsdf->f(wo, wi);
    if (!f.IsBlack() && visibility.Unoccluded(scene))
        L += f * Li * AbsDot(wi, n) *
             visibility.Transmittance(scene, renderer,
                                      sample, rng, arena) / pdf;
}
```

This integrator also handles light scattered by perfectly specular surfaces like mirrors or glass. It is fairly simple to use properties of mirrors to find the reflected directions (Figure 1.21), and to use Snell's law to find the transmitted directions (Section 8.2). The integrator can then recursively follow the appropriate ray in the new direction and add its contribution to the reflected radiance at the point originally seen from the camera. The computation of the effect of specular reflection and transmission is handled in separate utility functions so these functions can easily be reused by other `Integrators`.

AbsDot() 61
BSDF::f() 481
Integrator 740
Intersection::rayEpsilon 186
Light::Sample_L() 608
LightSample 710
Scene::lights 23
Spectrum 263
Spectrum::IsBlack() 265
Vector 57
VisibilityTester 608
VisibilityTester::
Transmittance() 609
VisibilityTester::
Unoccluded() 609

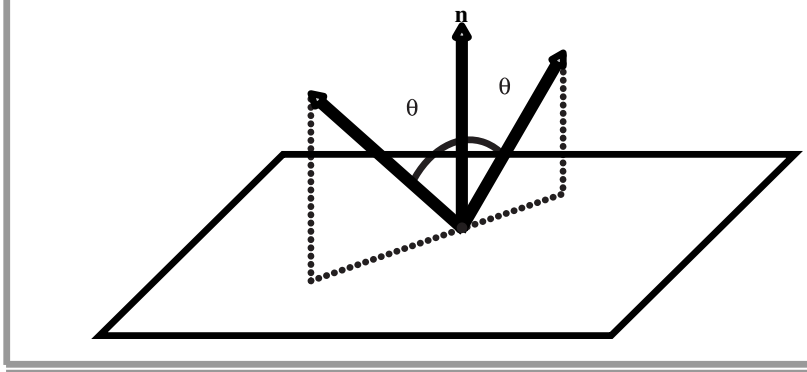

Figure 1.21: Reflected rays due to perfect specular reflection make the same angle with the surface normal as the incident ray.

⟨*Trace rays for specular reflection and refraction*⟩ ≡ 43, 789
```
L += SpecularReflect(ray, bsdf, rng, isect, renderer, scene, sample,
                     arena);
L += SpecularTransmit(ray, bsdf, rng, isect, renderer, scene, sample,
                      arena);
```

In the `SpecularReflect()` and `SpecularTransmit()` functions, the `BSDF::Samplef()` method returns an incident ray direction for a given outgoing direction and a given mode of light scattering. This method is one of the foundations of the Monte Carlo light transport algorithms that will be the subject of the last few chapters of this book. Here, we will use it to find only outgoing directions corresponding to perfect specular reflection or refraction, using flags to indicate to `BSDF::Sample_f()` that other types of reflection should be ignored. Although `BSDF::Sample_f()` can sample random directions leaving the surface for probabilistic integration algorithms, the randomness is constrained to be consistent with the `BSDF`'s scattering properties. In the case of perfect specular reflection or refraction, only one direction is possible, so there is no randomness at all.

The calls to `BSDF::Sample_f()` in these functions initialize `wi` with the chosen direction and return the BSDF's value for the directions (ω_o, ω_i). If the value of the BSDF is nonzero, the integrator uses the `Renderer::Li()` method to get the incoming radiance along ω_i, which in this case will in turn cause the `WhittedIntegrator::Li()` method to be called again. To compute the cosine term of the reflection integral, the integrator calls the `AbsDot()` function, which returns the absolute value of the dot product between two vectors. If the vectors are normalized, as both `wi` and `n` are here, this is equal to the absolute value of the cosine of the angle between them (Section 2.2.3).

Finally, in order to use ray differentials to antialias textures that are seen in reflections or refractions, it is necessary to know how reflection and transmission affect the screen-space footprint of rays. The fragments that compute the ray differentials for these rays are defined later, in Section 10.1.3.

⟨*Integrator Utility Functions*⟩ ≡
```
Spectrum SpecularReflect(const RayDifferential &ray, BSDF *bsdf,
        RNG &rng, const Intersection &isect, const Renderer *renderer,
        const Scene *scene, const Sample *sample, MemoryArena &arena) {
    Vector wo = -ray.d, wi;
    float pdf;
    const Point &p = bsdf->dgShading.p;
    const Normal &n = bsdf->dgShading.nn;
    Spectrum f = bsdf->Sample_f(wo, &wi, BSDFSample(rng), &pdf,
                                BxDFType(BSDF_REFLECTION | BSDF_SPECULAR));
    Spectrum L = 0.f;
    if (pdf > 0.f && !f.IsBlack() && AbsDot(wi, n) != 0.f) {
        ⟨Compute ray differential rd for specular reflection 512⟩
        Spectrum Li = renderer->Li(scene, rd, sample, rng, arena);
        L = f * Li * AbsDot(wi, n) / pdf;
    }
    return L;
}
```

AbsDot() 61
BSDF 478
BSDF::dgShading 479
BSDF::Sample_f() 706
BSDFSample 705
BSDF_REFLECTION 428
BSDF_SPECULAR 428
BxDFType 428
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
Intersection 186
MemoryArena 1015
Normal 65
Point 63
Ray::d 67
RayDifferential 69
Renderer 24
Renderer::Li() 25
RNG 1003
Sample 343
Scene 22
Spectrum 263
Spectrum::IsBlack() 265
SpecularReflect() 46
SpecularTransmit() 47
Vector 57
WhittedIntegrator::Li() 42

The SpecularTransmit() function is essentially the same as SpecularReflect(), but just requests the BSDF_TRANSMISSION specular component of the BSDF, if any, rather than the BSDF_REFLECTION component used by SpecularReflect(). We therefore won't include its implementation in the text of the book here.

1.4 HOW TO PROCEED THROUGH THIS BOOK

We have written this book assuming it will be read in roughly front-to-back order. We have tried to minimize the number of forward references to ideas and interfaces that haven't yet been introduced, but do assume that the reader is acquainted with the previous content at any particular point in the text. However, some sections go into depth about advanced topics that some readers may wish to skip over (particularly on first reading); each advanced section is identified by an asterisk in its title.

Because of the modular nature of the system, the main requirement is that the reader be familiar with the low-level classes like Point, Ray, and Spectrum; the interfaces defined by the abstract base classes listed in Table 1.1; and the main rendering loop in SamplerRenderer::Render(). Given that knowledge, for example, the reader who doesn't care about precisely how a camera model based on a perspective projection matrix maps samples to rays can skip over the implementation of that camera and can just remember that the Camera::GenerateRayDifferential() method somehow turns a Sample into a RayDifferential.

The rest of this book is divided into four main parts of a few chapters each. First, Chapters 2 through 4 define the main geometric functionality in the system. Chapter 2 has the low-level classes like Point, Ray, and BBox. Chapter 3 defines the Shape interface, gives implementations of a number of shapes, and shows how to perform ray-shape intersection tests. Chapter 4 has the implementations of the acceleration structures for speeding up ray tracing by avoiding tests with primitives that a ray can be shown to definitely not intersect.

The second part covers the image formation process. First, Chapter 5 introduces the physical units used to measure light, and the Spectrum class that represents wavelength-varying distributions (i.e., color). Chapter 6 defines the Camera interface and has a few different camera implementations. The Sampler classes that place samples on the image plane are the topic of Chapter 7, and the overall process of turning radiance values on the film into images suitable for display is explained in Section 7.8.

The third part of the book is about light and how it scatters from surfaces and participating media. Chapter 8 includes a set of building-block classes that define a variety of types of reflection from surfaces. Materials, described in Chapter 9, use these reflection functions to implement a number of different surface materials, such as plastic, glass, and metal. Chapter 10 introduces texture, which describes variation in material properties (color, roughness, etc.) over surfaces, and Chapter 11 has the abstractions that describe how light is scattered and absorbed in participating media. Finally, Chapter 12 has the interface for light sources and light source implementations.

The last part brings all of the ideas from the rest of the book together to implement a number of light transport algorithms. Chapters 13 and 14 introduce the theory of

BBox 70
BSDF_REFLECTION 428
BSDF_TRANSMISSION 428
Camera 302
Camera::
GenerateRayDifferential() 303
Point 63
Ray 66
RayDifferential 69
Sample 343
Sampler 340
SamplerRenderer::Render() 27
Shape 108
Spectrum 263

Monte Carlo integration, a statistical technique for estimating the value of complex integrals, and have low-level routines for applying Monte Carlo to illumination and light scattering. The surface and volume integrators in Chapters 15 and 16 use Monte Carlo integration to compute more accurate approximations of the light transport equation than the `WhittedIntegrator`, using techniques like path tracing, irradiance caching, and photon mapping. Chapter 17 shows a number of ways to use `pbrt` for precomputing data to use for later rendering; this is an increasingly important topic as researchers continue to develop new techinques for efficiently re-rendering scenes for interactive applications using data computed offline with rendering systems like `pbrt`.

Finally, the last chapter of the book provides a brief retrospective and discussion of system design decisions along with a number of suggestions for more far-reaching projects than those in the exercises. Appendices describe utility functions and details of how the scene description is created as the input file is parsed.

1.4.1 THE EXERCISES

At the end of each chapter you will find exercises related to the material covered in that chapter. Each exercise is marked as one of three levels of difficulty:

1. An exercise that should take only an hour or two
2. A reading and/or implementation task that would be suitable for a course assignment and should take between 10 and 20 hours of work
3. A suggested final project for a course that will likely take 40 hours or more to complete

1.5 USING AND UNDERSTANDING THE CODE

We have written `pbrt` in C++. However, we have used only a subset of the language, both to make the code easy to understand, as well as to maximize the system's portability. In particular, we have avoided multiple inheritance and run-time exception handling and have used only a small subset of C++'s extensive standard library.

We will occasionally omit short sections of `pbrt`'s source code from this document. For example, when there are a number of cases to be handled, all with nearly identical code, we will present one case and note that the code for the remaining cases has been omitted from the text. Of course, all the omitted code can be found in the `pbrt` source code distribution.

1.5.1 POINTER OR REFERENCE?

C++ provides two different mechanisms for passing the address of a data structure to a function: pointers and references. If a function argument is not intended as an output variable, either can be used to save the expense of passing the entire structure on the stack. By convention, `pbrt` uses pointers when the function argument will be modified in some way, and `const` references when it won't. One important exception to this rule is that we will always use a pointer when we want to be able to pass `NULL` to indicate that a parameter is not available or should not be used.

WhittedIntegrator 42

1.5.2 CODE OPTIMIZATION

We have tried to make `pbrt` efficient through the use of well-chosen algorithms rather than through local micro-optimizations. However, we have applied some local optimizations to the parts of `pbrt` that account for the most execution time, as long as doing so didn't make the code confusing. There are two main local optimization principles used throughout the code:

- On current CPU architectures, the slowest mathematical operations are divides, square roots, and trigonometric functions. Addition, subtraction, and multiplication are generally 10 to 50 times faster than those operations. Reducing the number of slow mathematical operations can help performance substantially; for example, instead of repeatedly dividing by some value v, we will tend to precompute the reciprocal $1/v$ and multiply by that instead.
- The speed of CPUs continues to grow more quickly than the speed at which data can be loaded from main memory into the CPU. This means that waiting for values to be fetched from memory can be a major performance limitation. Organizing algorithms and data structures in ways that give good performance from memory caches can speed up program execution much more than reducing the total number of instructions executed. Section A.5 in Appendix A discusses general principles for memory-efficient programming; these ideas are mostly applied in the ray intersection acceleration structures of Chapter 4 and the image map representation in Section 10.4.2, although they influence many of the design decisions throughout the system.

1.5.3 THE BOOK WEB SITE

We have created a companion Web site for this book, located at *www.pbrt.org*. The Web site includes the system's complete source code, additional modules, example scenes, and errata. It also hosts contributed `pbrt` modules and scene file conversion tools from readers.

All readers are encouraged to visit the Web site and follow the instructions to subscribe to the *pbrt-announce* mailing list; information about how to join it is available at *pbrt.org/lists.php*. We will occasionally send announcements of software updates to this list, so readers can always have the latest version of `pbrt` and all the latest modules. The Web site also has links to an online discussion forum for `pbrt` and a bug-tracking system.

1.5.4 EXTENDING THE SYSTEM

One of our goals in writing this book and building the `pbrt` system was to make it easier for developers and researchers to experiment with new (or old!) ideas in rendering. One of the great joys in computer graphics is writing new software that makes a new image; even small changes to the system can be fun to experiment with. The exercises throughout the book suggest many changes to make to the system, ranging from small tweaks to major open-ended research projects. Section B.4 in Appendix B has more information about the mechanics of how to extend the system.

1.5.5 BUGS

Although we have made every effort to make pbrt as correct as possible through extensive testing, it is inevitable that some bugs are still present.

If you believe you have found a bug in the system, please do the following:

1. Reproduce the bug with an unmodified copy of the latest version of pbrt.
2. Check the online discussion forum and the bug-tracking system at *www.pbrt.org*. Your issue may be a known bug, or it may be a commonly misunderstood feature.
3. Try to find make the simplest possible test case that demonstrates the bug. Many bugs can be demonstrated by input files that are just a few lines long, and debugging is much easier with a simple scene than a complex one.
4. Submit a detailed bug report using our online bug-tracking system. Make sure that you include the short input file that demonstrates the bug and a detailed description of why you think pbrt is not behaving correctly on your input. If you can provide a patch to the code that fixes the bug, all the better!

We will periodically release updated versions of pbrt with bug fixes and minor enhancements. However, we will not make major changes to the updated versions of the pbrt source code distributed from the book's Web site so that the system described here in the book doesn't diverge excessively from the updated versions of pbrt.

FURTHER READING

In a seminal early paper, Arthur Appel (1968) first described the basic idea of ray tracing to solve the hidden surface problem and to compute shadows in polygonal scenes. Goldstein and Nagel (1971) later showed how ray tracing could be used to render scenes with quadric surfaces. Kay and Greenberg (1979) described a ray-tracing approach to rendering transparency, and Whitted's seminal *CACM* article described the general recursive ray-tracing algorithm we have outlined in this chapter, accurately simulating reflection and refraction from specular surfaces and shadows from point light sources (Whitted 1980). Heckbert (1987) was the first to explore realistic rendering of dessert.

Notable early books on physically based rendering and image synthesis include Cohen and Wallace's *Radiosity and Realistic Image Synthesis* (Cohen and Wallace 1993), Sillion and Puech's *Radiosity and Global Illumination* (Sillion and Puech 1994), and Ashdown's *Radiosity: A Programmer's Perspective* (Ashdown 1994), all of which primarily describe the finite-element radiosity method. Glassner's *Principles of Digital Image Synthesis* (Glassner 1995) is an encyclopedic two-volume summary of theoretical foundations for realistic rendering. Hall's *Illumination and Color in Computer Generated Imagery* (Hall 1989) is one of the first books to present rendering in a physically based framework. Dutré et al.'s *Advanced Global Illumination* has extensive and up-to-date coverage of these topics (Dutré, Bala, and Bekaert 2006).

Greenberg et al. (1997) made a strong argument for a physically accurate rendering based on measurements of the material properties of real-world objects and on deep understanding of the human visual system. These ideas have served as a basis for much of the rendering research done at Cornell over the past two decades.

In a paper on ray-tracing system design, Kirk and Arvo (1988) suggested many principles that have now become classic in renderer design. Their renderer was implemented as a core kernel that encapsulated the basic rendering algorithms and interacted with primitives and shading routines via a carefully constructed object-oriented interface. This approach made it easy to extend the system with new primitives and acceleration methods. `pbrt`'s design is based on these ideas.

Another good reference on ray tracer design is *Introduction to Ray Tracing* (Glassner 1989a), which describes the state of the art in ray tracing at that time and has a chapter by Heckbert that sketches the design of a basic ray tracer. More recently, Shirley and Morley's *Realistic Ray Tracing* gives an easy-to-understand introduction to ray tracing and includes the complete source code to a basic ray tracer (Shirley and Morley 2008). Suffern's book also provides a gentle introduction to ray tracing (Suffern 2007).

Researchers at Cornell University have developed a rendering testbed over many years; its design and overall structure were described by Trumbore, Lytle, and Greenberg (1993). Its predecessor was described by Hall and Greenberg (1983). This system is a loosely coupled set of modules and libraries, each designed to handle a single task (ray–object intersection acceleration, image storage, etc.) and written in a way that makes it easy to combine appropriate modules to investigate and develop new rendering algorithms. This testbed has been quite successful, serving as the foundation for much of the rendering research done at Cornell.

Many papers have been written that describe the design and implementation of other rendering systems, including renderers for entertainment and artistic applications. The Reyes architecture, which forms the basis for Pixar's RenderMan® renderer, was first described by Cook, Carpenter, and Catmull (1987), and a number of improvements to the original algorithm have been summarized by Apodaca and Gritz (2000). Gritz and Hahn (1996) described the BMRT ray tracer. The renderer in the Maya modeling and animation system was described by Sung et al. (1998), and some of the internal structure of the *mental ray* renderer is described in Driemeyer and Herken's book on its API (Driemeyer and Herken 2002). The design of the high-performance *Manta* interactive ray tracer is described by Bigler et al. (2006).

Another category of renderer focuses on physically based rendering, like `pbrt`. One of the first renderers based fundamentally on physical quantities is called *Radiance*, and it has been used widely in lighting simulation applications. Ward described its design and history in a paper and a book (Ward 1994b; Larson and Shakespeare 1998). *Radiance* is designed in the UNIX style, as a set of interacting programs, each handling a different part of the rendering process. This general type of rendering architecture was first described by Duff (1985).

Glassner's *Spectrum* rendering architecture also focuses on physically based rendering (Glassner 1993), approached through a signal-processing-based formulation of the problem. It is an extensible system built with a plug-in architecture; `pbrt`'s approach of using parameter/value lists for initializing implementations of the main abstract interfaces is similar to *Spectrum*'s. One notable feature of *Spectrum* is that all parameters that describe the scene can be functions of time.

Slusallek and Seidel described the architecture of the *Vision* rendering system, which is also physically based and designed to support a wide variety of light transport algorithms (Slusallek and Seidel 1995, 1996; Slusallek 1996). In particular, it has the ambitious goal of supporting both Monte Carlo and finite-element-based light transport algorithms.

The source code to `pbrt` is licensed under the GNU General Public License; this has made it possible for other developers to use `pbrt` code as a basis for their efforts. *LuxRender*, available from *www.luxrender.net*, is a physically based renderer built using `pbrt` as a starting point; it offers a number of additional features and has a rich set of scene export plugins for modeling systems.

The source code to a number of other ray tracers and renderers is available on the Web. Because this is a rapidly changing field, we maintain a set of links to interesting open source rendering systems online at *www.pbrt.org/links.php*.

A good introduction to the C++ programming language and C++ standard library is the third edition of Stroustrup's *The C++ Programming Language* (Stroustrup 1997).

EXERCISE

1 1.1 A good way to gain an understanding of `pbrt` is to follow the process of computing the radiance value for a single ray in a debugger. Build a version of `pbrt` with debugging symbols and set up your debugger to run `pbrt` with the `killeroo-simple.pbrt` scene from the `scenes/` directory. Set breakpoints in the `SamplerRenderer::Render()` and `SamplerRendererTask::Run()` methods and trace through the process of how a ray is generated, how its radiance value is computed, and how its contribution is added to the image. The first time you do this, you may want to specify that only a single thread of execution should be used by providing `--ncores 1` as command-line arguments to `pbrt`; doing so ensures that all computation is done in the main processing thread, which may make it easier to understand what is going on, depending on how easy your debugger makes it to step through the program when it is running multiple threads.

As you gain more understanding about the details of the system later in the book, repeat this process and trace through particular parts of the system more carefully.

`SamplerRenderer::Render()` 27
`SamplerRendererTask::Run()` 30