CHAPTER ELEVEN

★ 11 VOLUME SCATTERING

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

This chapter first describes the basic physical processes that affect the radiance along rays passing through participating media. It then introduces the `VolumeRegion` base class, an interface for modeling different types of media. Like a `BSDF`, the volume description characterizes how light is scattered at individual points. In order to determine the global effect on the distribution of light in the scene, `VolumeIntegrators` are necessary; they are described in Chapter 16. The chapter concludes with the abstraction that represents the subsurface scattering properties of objects, the `BSSRDF`, as well a number of `Materials` for translucent objects.

11.1 VOLUME SCATTERING PROCESSES

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

- *Absorption*—the reduction in radiance due to the conversion of light to another form of energy, such as heat
- *Emission*—energy that is added to the environment from luminous particles

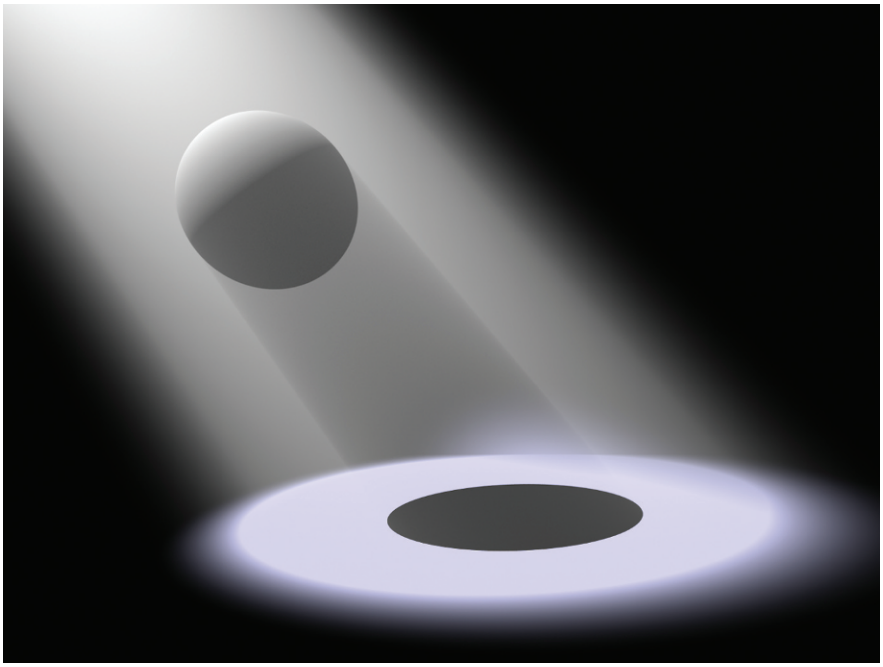

Figure 11.1: Spotlight through Fog. Light scattering from particles in the medium back toward the camera makes the spotlight's illumination visible even in pixels where there are no visible surfaces that reflect it. The sphere blocks light, casting a volumetric shadow in the region beneath it.

- *Scattering*—how light heading in one direction is scattered to other directions due to collisions with particles

The characteristics of all of these properties may be *homogeneous* or *inhomogeneous*. Homogeneous properties are constant throughout a given spatial extent, while inhomogeneous properties may vary throughout space. Figure 11.1 shows a simple example of volume scattering, where a spotlight shining through a participating medium illuminates the medium and and casts a volumetric shadow.

11.1.1 ABSORPTION

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

Absorption is described by the medium's *absorption cross section*, σ_a, which is the probability density that light is absorbed per unit distance traveled in the medium. In general, the absorption cross section may vary with both position p and direction ω, although it is normally just a function of position. It is usually also a spectrally varying quantity. The

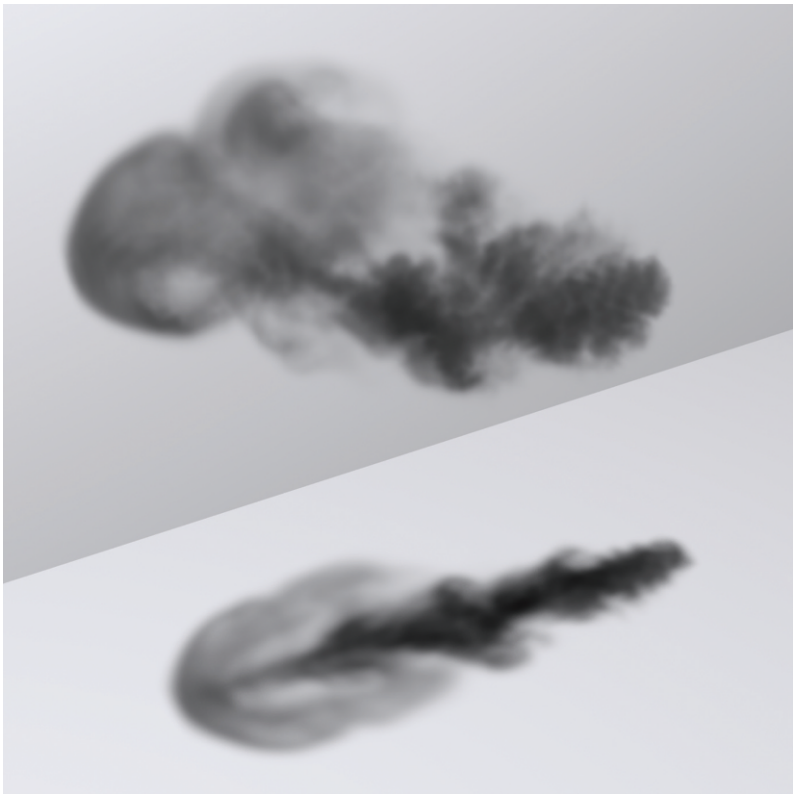

Figure 11.2: If a participating medium primarily absorbs light passing through it, it will have a dark and smoky appearance, as shown here. *(Smoke simulation data courtesy of Duc Nguyen and Ron Fedkiw.)*

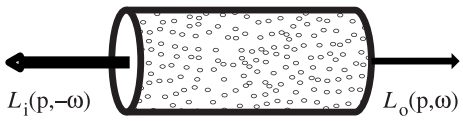

Figure 11.3: Absorption reduces the amount of radiance along a ray through a participating medium. Consider a ray carrying incident radiance at a point p from direction $-\omega$. If the ray passes through a differential cylinder filled with absorbing particles, the change in radiance due to absorption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

units of σ_a are reciprocal distance (m^{-1}). This means that σ_a can take on any positive value; it is not required to be between zero and one, for instance.

Figure 11.3 shows the effect of absorption along a very short portion of a ray. Some amount of radiance $L_i(p, -\omega)$ is arriving at point p, and we'd like to find the exitant

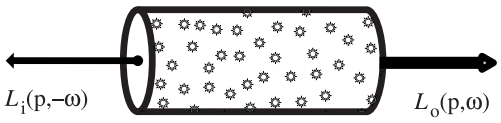

Figure 11.4: The volume emission function $L_{ve}(p, \omega)$ gives the change in radiance along a ray as it passes through a differential volume of emissive particles. The change in radiance per differential distance is $dL = L_{ve}dt$.

radiance $L_o(p, \omega)$ after absorption in the differential volume. This change in radiance along the differential ray length dt is described by the differential equation

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

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

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

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

11.1.2 EMISSION

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

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

$$dL_o(p, \omega) = L_{ve}(p, \omega)\, dt.$$

Note that this equation is based on the assumption that the emitted light L_{ve} is not dependent on the incoming light L_i. This is always true under the linear optics assumptions that `pbrt` is based on.

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

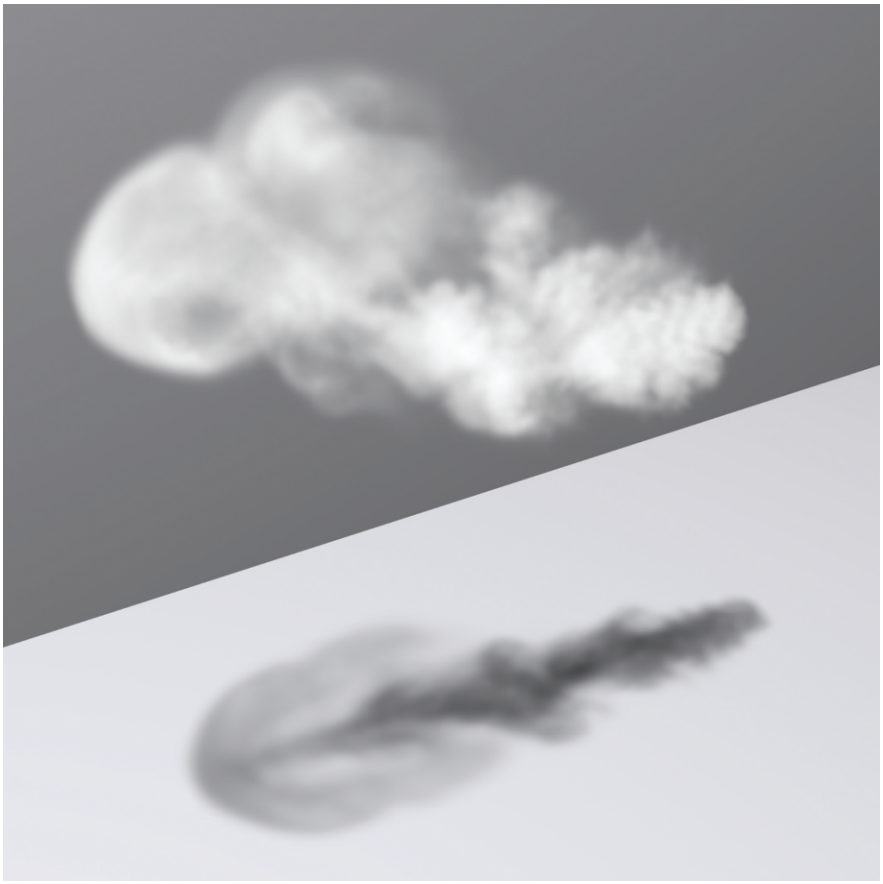

Figure 11.5: A Participating Medium Where the Dominant Volumetric Effect Is Emission. Although the medium still absorbs light, still casting a shadow on the ground and obscuring the wall behind it, emission in the volume increases radiance along rays passing through it, making the cloud brighter than the wall behind it.

11.1.3 OUT-SCATTERING AND EXTINCTION

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

The probability of an out-scattering event occurring per unit distance is given by the scattering coefficient, σ_s. As with the attenuation coefficient, the reduction in radiance

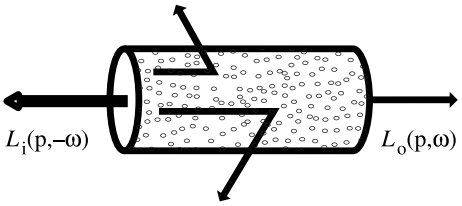

Figure 11.6: Like absorption, out-scattering also reduces the radiance along a ray. Light that hits particles may be scattered in another direction such that the radiance exiting the region in the original direction is reduced.

along a differential length dt due to out-scattering is given by

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

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

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

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

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

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

$$T_r(p \to p') = e^{-\int_0^d \sigma_t(p+t\omega,\omega)\, dt}$$

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

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

This idea is illustrated in Figure 11.7.

Two useful properties of beam transmittance are that transmittance from a point to itself is one ($T_r(p \to p) = 1$), and in a vacuum $T_r(p \to p') = 1$ for all p′. Another important property, true in all media, is that transmittance is multiplicative along points on a ray:

$$T_r(p \to p'') = T_r(p \to p')T_r(p' \to p''),$$

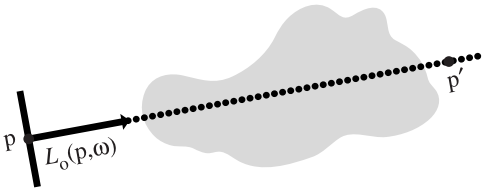

Figure 11.7: The beam transmittance $T_r(\mathrm{p} \to \mathrm{p}')$ gives the fraction of light transmitted from one point to another, accounting for absorption and out-scattering, but ignoring emission and in-scattering. Given exitant radiance at a point p in direction ω (e.g., reflected radiance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

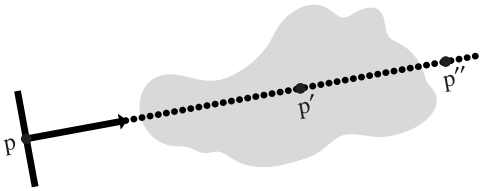

Figure 11.8: A useful property of beam transmittance is that it is multiplicative: the transmittance between points p and p″ on a ray like the one shown here is equal to the transmittance from p to p′ times the transmittance from p′ to p″ for all points p′ between p and p″.

for all points p′ between p and p″ (Figure 11.8). This property is important for volume scattering implementations, since it makes it possible to incrementally compute transmittance at many points along a ray by computing the product of each previously computed transmittance with the transmittance for its next segment.

The negated exponent in T_r is called the *optical thickness* between the two points. It is denoted by the symbol τ:

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

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

$$T_r(\mathrm{p} \to \mathrm{p}') = \mathrm{e}^{-\sigma_t d}. \tag{11.1}$$

11.1.4 IN-SCATTERING

While out-scattering reduces radiance along a ray due to scattering in different directions, *in-scattering* accounts for increased radiance due to scattering from other directions

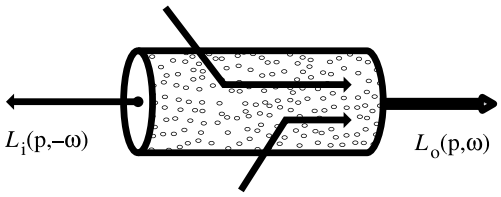

Figure 11.9: In-scattering accounts for the increase in radiance along a ray due to scattering of light from other directions. Radiance from outside the differential volume is scattered along the direction of the ray and added to the incoming radiance.

(Figure 11.9). Figure 11.10 shows the effect of in-scattering with the smoke data set. Note that the smoke appears much thicker than when absorption or emission was the dominant volumetric effect.

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

$$\int_{S^2} p(\omega \to \omega')\, d\omega' = 1 \tag{11.2}$$

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

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

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

It accounts for both volume emission and in-scattering:

$$S(p, \omega) = L_{ve}(p, \omega) + \sigma_s(p, \omega) \int_{S^2} p(p, -\omega' \to \omega)\, L_i(p, \omega')\, d\omega'.$$

The in-scattering portion of the source term is the product of the scattering probability per unit distance, σ_s, and the amount of added radiance at a point, which is given by the spherical integral of the product of incident radiance and the phase function. Note that the source term is very similar to the scattering equation, Equation (5.8); the main difference is that there is no cosine term since the phase function operates on radiance rather than differential irradiance.

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

11.2 PHASE FUNCTIONS

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

In most naturally occurring media, the phase function is a 1D function of only the angle θ between the two directions ω and ω'; such media are called *isotropic*, and these phase functions are often written as $p(\cos\theta)$. In exotic media, such as those with a crystalline structure, the phase function is a 4D function of the two directions. In addition to being normalized, an important property of naturally occurring phase functions is that they are *reciprocal*: the two directions can be interchanged and the phase function's value remains unchanged. Note that isotropic phase functions are trivially reciprocal because $\cos(-\theta) = \cos(\theta)$.

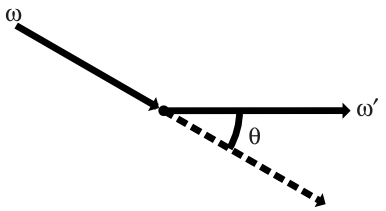

Figure 11.11: Phase functions are written with the convention that the incident direction points toward the point where scattering happens and the outgoing direction ω' points away from it. This is a different convention than was used for BSDFs. The angle between them is denoted by θ.

In a slightly confusing overloading of terminology, phase functions themselves can be isotropic or anisotropic as well. An isotropic phase function describes equal scattering in all directions and is thus independent of either of the two directions. Because phase functions are normalized, there is only one such function, and it must be the constant $1/4\pi$:

$$p_{\text{isotropic}}(\omega \to \omega') = \frac{1}{4\pi}.$$

⟨*Volume Scattering Definitions*⟩ ≡

```
float PhaseIsotropic(const Vector &, const Vector &) {
    return 1.f / (4.f * M_PI);
}
```

All of the anisotropic phase functions in the remainder of this section describe isotropic media and are thus defined in terms of the angle between the two directions (Figure 11.11). Phase functions use a different convention for the direction of the vectors at a scattering event than was used for scattering at a surface, where both vectors faced away from the surface. For phase function implementations in `pbrt`, the incident direction points toward the scattering point and the outgoing direction points away from it. This matches the usual convention for phase functions.

`pbrt` includes implementations of phase functions that model Rayleigh scattering and Mie scattering. The Rayleigh model describes scattering from very small particles such as the molecules in the Earth's atmosphere. If the particles have radii that are smaller than the wavelength of light λ, ($r/\lambda < 0.05$), the Rayleigh model accurately describes the distribution of scattered light. Wavelength-dependent Rayleigh scattering is the reason that the sky is blue and sunsets are red. Mie scattering is based on a more general theory; it is derived from Maxwell's equations and can describe scattering from a wider range of particle sizes. For example, it is a good model for scattering in the atmosphere due to water droplets and fog.

M_PI 1002
Vector 57

We won't include the implementations of these phase functions in the text here. They are straightforward transcriptions of models from the research literature.

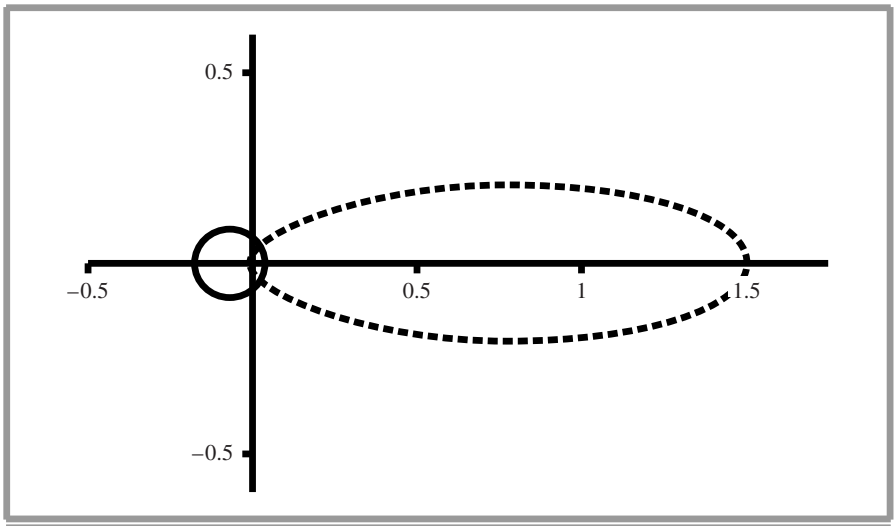

Figure 11.12: Plots of the Henyey–Greenstein Phase Function for Asymmetry g Parameters $-.35$ and $.67$. Negative g values (solid line) describe phase functions that primarily scatter light back in the incident direction, and positive g values (dashed line) describe phase functions that primarily scatter light forward in the direction it was already traveling.

⟨*Volume Scattering Declarations*⟩ ≡
```
float PhaseRayleigh(const Vector &w, const Vector &wp);
float PhaseMieHazy(const Vector &w, const Vector &wp);
float PhaseMieMurky(const Vector &w, const Vector &wp);
```

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

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

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

⟨*Volume Scattering Definitions*⟩ +≡
```
float PhaseHG(const Vector &w, const Vector &wp, float g) {
    float costheta = Dot(w, wp);
    return 1.f / (4.f * M_PI) *
        (1.f - g*g) / powf(1.f + g*g - 2.f * g * costheta, 1.5f);
}
```

M_PI 1002
Vector 57

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

$$g = \int_{S^2} p(\omega \to \omega')(\omega \cdot \omega') \, d\omega' = 2\pi \int_0^\pi p(\cos\theta) \cos\theta \sin\theta \, d\theta.$$

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

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

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

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

where the weights w_i sum to one to maintain normalization.

Blasi, Le Saëc, and Schlick (1993) developed an efficient approximation to the Henyey–Greenstein function that has been widely used in computer graphics due to its computational efficiency. It avoids the expensive `powf()` function:

$$p_{\text{Schlick}}(\cos\theta) = \frac{1}{4\pi} \frac{1 - k^2}{(1 - k\cos\theta)^2}.$$

The k parameter has a similar effect on the distribution to the g term of the Henyey–Greenstein model, where -1 corresponds to total back-scattering, 0 corresponds to isotropic scattering, and 1 corresponds to total forward-scattering. For immediate values, we have found that the polynomial equation

$$k = 1.55g - .55g^3$$

gives an accurate correspondence between k and g values. Figure 11.13 shows a plot of the Schlick model and the Henyey–Greenstein model.

⟨*Volume Scattering Definitions*⟩ +≡

```
float PhaseSchlick(const Vector &w, const Vector &wp, float g) {
    float k = 1.55f * g - .55f * g * g * g;
    float kcostheta = k * Dot(w, wp);
    return 1.f / (4.f * M_PI) *
        (1.f - k*k) / ((1.f - kcostheta) * (1.f - kcostheta));
}
```

M_PI 1002
Vector 57

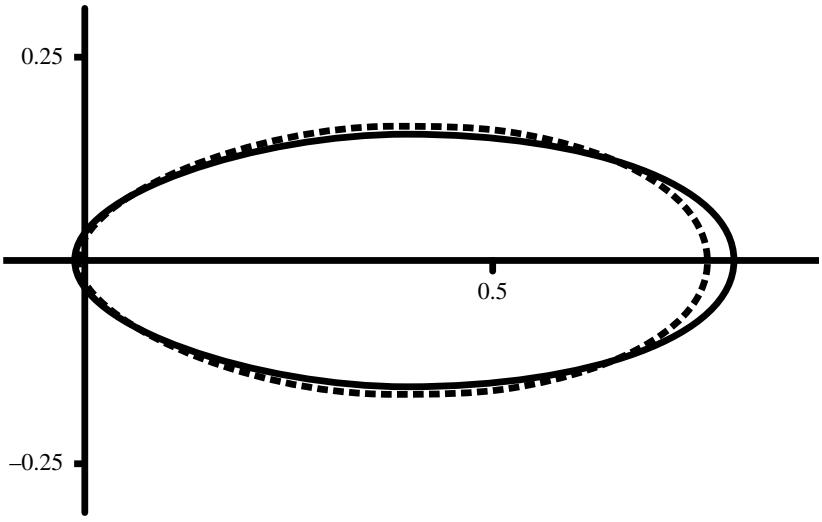

Figure 11.13: Plot of the Schlick phase function with $k = .8112$ (dashed lines) superimposed with plot of the Henyey–Greenstein phase function (solid lines) with $g = .6$. The Schlick model closely matches Henyey–Greenstein and is less computationally expensive to evaluate.

11.3 VOLUME INTERFACE AND HOMOGENEOUS MEDIA

The key abstraction for describing volume scattering in `pbrt` is the abstract `VolumeRegion` class—the interface to describe volume scattering in a region of the scene. Multiple `VolumeRegions` of different types can be used to describe different kinds of scattering in different parts of the scene. In this section, we will describe the basic interface, which is defined in the files `core/volume.h` and `core/volume.cpp`, as well as a few useful implementations of that interface, all of which are in the `volumes/` directory of the `pbrt` distribution.

⟨*Volume Scattering Declarations*⟩ +≡
```
class VolumeRegion {
public:
    ⟨VolumeRegion Interface 587⟩
};
```

All `VolumeRegions` must be able to compute their axis-aligned world space bounding box, which is returned by the `VolumeRegion::WorldBound()` method. As with `Shapes` and `Primitives`, this bound can be used to place `VolumeRegions` into acceleration structures.

BBox 70
VolumeRegion 587
VolumeRegion::WorldBound() 587

⟨*VolumeRegion Interface*⟩ ≡ 587
```
virtual BBox WorldBound() const = 0;
```

Because VolumeIntegrators need to know the parametric range of a world space ray that passes through a volume region, and because a world space bounding box may not tightly bound a particular VolumeRegion, a separate method VolumeRegion::IntersectP() returns the parametric t range of the segment that overlaps the volume, if any.

⟨*VolumeRegion Interface*⟩ +≡ 587
```
virtual bool IntersectP(const Ray &ray, float *t0, float *t1) const = 0;
```

This interface has four methods corresponding to the spatially varying scattering properties introduced earlier in this chapter. Given a world space point, direction, and time, VolumeRegion::sigma_a(), VolumeRegion::sigma_s(), and VolumeRegion::Lve() return the corresponding absorption, scattering, and emission properties. Given a pair of directions, the VolumeRegion::p() method returns the value of the phase function at the given point and the given time.

⟨*VolumeRegion Interface*⟩ +≡ 587
```
virtual Spectrum sigma_a(const Point &, const Vector &,
                         float time) const = 0;
virtual Spectrum sigma_s(const Point &, const Vector &,
                         float time) const = 0;
virtual Spectrum Lve(const Point &, const Vector &,
                     float time) const = 0;
virtual float p(const Point &, const Vector &,
                const Vector &, float time) const = 0;
```

For convenience, there is also a VolumeRegion::sigma_t() method that returns the attenuation coefficient at a point. A default implementation returns the sum of the σ_a and σ_s values, but most of the VolumeRegion implementations will override this method and compute σ_t directly. For VolumeRegions that need to do some amount of computation to find the values of σ_a and σ_s at a particular point, it's usually possible to avoid some duplicated work when σ_t is actually needed and to compute its value directly.

⟨*Volume Scattering Definitions*⟩ +≡
```
Spectrum VolumeRegion::sigma_t(const Point &p, const Vector &w,
                               float time) const {
    return sigma_a(p, w, time) + sigma_s(p, w, time);
}
```

Finally, the VolumeRegion::tau() method computes the volume's optical thickness from the point ray(ray.mint) to ray(ray.maxt). Some implementations, like the HomogeneousVolumeDensity in the next section, can compute this value exactly, while others use Monte Carlo integration to compute it. For the benefit of the Monte Carlo approach, this method takes two optional parameters, step and offset, that are ignored by implementations that compute this value in closed form. These Monte Carlo routines are defined in Section 14.7, and the meanings of these extra parameters are described there.

⟨*VolumeRegion Interface*⟩ +≡ 587
```
virtual Spectrum tau(const Ray &ray, float step = 1.f,
                     float offset = 0.5) const = 0;
```

HomogeneousVolumeDensity 589
Point 63
Ray 66
Spectrum 263
Vector 57
VolumeIntegrator 876
VolumeRegion 587
VolumeRegion::IntersectP() 588
VolumeRegion::Lve() 588
VolumeRegion::p() 588
VolumeRegion::sigma_a() 588
VolumeRegion::sigma_s() 588
VolumeRegion::sigma_t() 588
VolumeRegion::tau() 588

Figure 11.14: Dragon Model Inside a Homogeneous Volume with Uniform Scattering Properties throughout Its Spatial Extent.

11.3.1 HOMOGENEOUS VOLUMES

The simplest volume representation, `HomogeneousVolumeDensity`, describes a box-shaped region of space with homogeneous scattering properties. Values for σ_a, σ_s, the phase function's g value, and the amount of emission L_{ve} are passed to the constructor. In conjunction with a transformation from world space to the volume's object space and an axis-aligned object space bound, this suffices to describe the region's scattering properties and spatial extent. Its implementation is in `volumes/homogeneous.h` and `volumes/homogeneous.cpp`. Figure 11.14 shows an image of the dragon model inside a homogeneous volume.

⟨*HomogeneousVolumeDensity Declarations*⟩ ≡

```
class HomogeneousVolumeDensity : public VolumeRegion {
public:
    ⟨HomogeneousVolumeDensity Public Methods 590⟩
private:
    ⟨HomogeneousVolumeDensity Private Data 590⟩
};
```

HomogeneousVolumeDensity 589
VolumeRegion 587

The constructor, not shown here, initializes the member variables by copying the corresponding parameters.

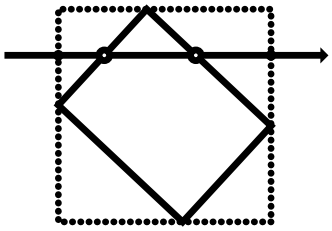

Figure 11.15: Volumes described by axis-aligned bounding boxes in the volume's object space (solid box) can compute a tighter bound on the parametric t range of a ray that overlaps the volume by transforming the ray into object space and computing the ray–box intersections there than if they find the intersections with the world space bound (dotted box) and an untransformed ray in world space. Here, the filled circles denote the world space intersections, and the open circles denote the object space intersections.

⟨*HomogeneousVolumeDensity Private Data*⟩ ≡ 589
```
Spectrum sig_a, sig_s, le;
float g;
BBox extent;
Transform WorldToVolume;
```

Because the bound is maintained internally in the volume's object space, it must be transformed to world space for the `WorldBound()` method.

⟨*HomogeneousVolumeDensity Public Methods*⟩ ≡ 589
```
BBox WorldBound() const {
    return Inverse(WorldToVolume)(extent);
}
```

If the region's world-to-object-space transformation includes a rotation such that the volume isn't axis aligned in world space, it is possible to compute a tighter segment of the ray–volume overlap by transforming the ray to the volume's object space and doing the overlap test there (Figure 11.15).

⟨*HomogeneousVolumeDensity Public Methods*⟩ +≡ 589
```
bool IntersectP(const Ray &r, float *t0, float *t1) const {
    Ray ray = WorldToVolume(r);
    return extent.IntersectP(ray, t0, t1);
}
```

The rest of the `VolumeRegion` interface methods are straightforward. Each one verifies that the given point is inside the region's extent and returns the appropriate value if so. The `sigma_a()` method illustrates the basic approach; the rest of the methods won't be included here.

BBox 70
BBox::IntersectP() 194
HomogeneousVolumeDensity::extent 590
HomogeneousVolumeDensity::WorldToVolume 590
Inverse() 1021
Ray 66
Spectrum 263
Transform 76
VolumeRegion 587

⟨*HomogeneousVolumeDensity Public Methods*⟩ +≡ 589
```
Spectrum sigma_a(const Point &p, const Vector &, float) const {
    return extent.Inside(WorldToVolume(p)) ? sig_a : 0.;
}
```

Because σ_a and σ_s are constant throughout the volume, the optical thickness that a ray passes through can be computed in closed form using Beer's law; see Equation (11.1).

⟨*HomogeneousVolumeDensity Public Methods*⟩ +≡ 589
```
Spectrum tau(const Ray &ray, float, float) const {
    float t0, t1;
    if (!IntersectP(ray, &t0, &t1)) return 0.;
    return Distance(ray(t0), ray(t1)) * (sig_a + sig_s);
}
```

11.4 VARYING-DENSITY VOLUMES

The rest of the volume representations in this chapter are based on the assumption that the underlying particles throughout the medium all have the same basic scattering properties, but their density is spatially varying in the medium. One consequence of this assumption is that it is possible to describe the volume scattering properties at a point as the product of the density at that point and some baseline value. For example, we might set the attenuation coefficient σ_t to have a base value of 0.2. In regions where the particle density was 1, a σ_t value of 0.2 would be returned. If the particle density were 3, however, a σ_t value of 0.6 would be the result.

In order to reduce duplicated code and allow the various representations to focus on different methods for defining the density of the particles, we will define a `DensityRegion` class that provides a new pure virtual method to obtain the particle density at a point. Volume representations can then inherit from `DensityRegion` and need not reimplement the substantial amount of shared logic that multiplies the density values by the base values at lookup points.

⟨*Volume Scattering Declarations*⟩ +≡
```
class DensityRegion : public VolumeRegion {
public:
    ⟨DensityRegion Public Methods 592⟩
protected:
    ⟨DensityRegion Protected Data 592⟩
};
```

The `DensityRegion` constructor takes the basic values of the scattering properties and stores them in corresponding member variables. Note that the interface specifies the volume-to-world transformation, but the class instead stores the world-to-volume transformation.

BBox::Inside() 72
DensityRegion 591
Distance() 65
HomogeneousVolumeDensity::extent 590
HomogeneousVolumeDensity::IntersectP() 590
HomogeneousVolumeDensity::sig_a 590
HomogeneousVolumeDensity::sig_s 590
HomogeneousVolumeDensity::WorldToVolume 590
Point 63
Ray 66
Spectrum 263
Vector 57
VolumeRegion 587

⟨*DensityRegion Public Methods*⟩ ≡ 591
```
DensityRegion(const Spectrum &sa, const Spectrum &ss, float gg,
              const Spectrum &emit, const Transform &VolumeToWorld)
    : sig_a(sa), sig_s(ss), le(emit), g(gg),
      WorldToVolume(Inverse(VolumeToWorld)) { }
```

⟨*DensityRegion Protected Data*⟩ ≡ 591
```
Spectrum sig_a, sig_s, le;
float g;
Transform WorldToVolume;
```

All `DensityRegion` implementations must implement the `DensityRegion::Density()` method, which returns the volume's density at the given point in object space. The density is used to scale the basic scattering parameters, so it must be nonnegative everywhere.

⟨*DensityRegion Public Methods*⟩ +≡ 591
```
virtual float Density(const Point &Pobj) const = 0;
```

The `DensityRegion::sigma_a()` method is illustrative of how a `DensityRegion` works; it scales `DensityRegion::sig_a` by the local density at the point. The other `VolumeRegion` methods are similar and not shown here.

⟨*DensityRegion Public Methods*⟩ +≡ 591
```
Spectrum sigma_a(const Point &p, const Vector &, float) const {
    return Density(WorldToVolume(p)) * sig_a;
}
```

One exception is the `DensityRegion::p()` method, which does not scale the phase function's value by the local density. Variations in the amount of scattering from point to point are already accounted for by the scaled σ_s values.

⟨*DensityRegion Public Methods*⟩ +≡ 591
```
float p(const Point &p, const Vector &w, const Vector &wp, float) const {
    return PhaseHG(w, wp, g);
}
```

The `DensityRegion` cannot implement the `VolumeRegion::tau()` method in closed form, since this method depends on global knowledge of the shape of the `VolumeRegion` as well as the density distribution throughout it. However, it can be implemented with Monte Carlo, as is done in Section 15.7.2.

DensityRegion 591
DensityRegion::Density() 592
DensityRegion::g 592
DensityRegion::p() 592
DensityRegion::sigma_a() 592
DensityRegion::sig_a 592
DensityRegion::WorldToVolume 592
ImageTexture 524
Inverse() 1021
PhaseHG() 585
Point 63
Spectrum 263
Transform 76
Vector 57
VolumeGridDensity 593
VolumeRegion 587
VolumeRegion::tau() 588

11.4.1 3D GRIDS

The `VolumeGridDensity` class stores densities at a regular 3D grid of positions, similar to the way that the `ImageTexture` represents images with a 2D grid of samples. These samples are interpolated to compute the density at positions between the sample points. The constructor takes a 3D array of user-supplied density values, thus allowing a variety of sources of data (physical simulation, CT scan, etc.). The smoke data set rendered in Figures 11.2, 11.5, and 11.10 is represented with a `VolumeGridDensity`. Because this class is a subclass of `DensityRegion`, the user also supplies baseline values of σ_a,

σ_s, L_{ve}, and g to the constructor. The implementation of the VolumeGridDensity is in volumes/volumegrid.h and volumes/volumegrid.cpp.

⟨*VolumeGridDensity Declarations*⟩ ≡
```
class VolumeGridDensity : public DensityRegion {
public:
    ⟨VolumeGridDensity Public Methods 593⟩
private:
    ⟨VolumeGridDensity Private Data 593⟩
};
```

The constructor does the usual initialization of the basic scattering properties, stores an object space bounding box for the region, and makes a local copy of the density values.

⟨*VolumeGridDensity Public Methods*⟩ ≡ 593
```
VolumeGridDensity(const Spectrum &sa, const Spectrum &ss, float gg,
        const Spectrum &emit, const BBox &e, const Transform &v2w,
        int x, int y, int z, const float *d)
    : DensityRegion(sa, ss, gg, emit, v2w), nx(x), ny(y), nz(z), extent(e) {
    density = new float[nx*ny*nz];
    memcpy(density, d, nx*ny*nz*sizeof(float));
}
```

⟨*VolumeGridDensity Private Data*⟩ ≡ 593
```
float *density;
const int nx, ny, nz;
const BBox extent;
```

The implementations of the WorldBound() and IntersectP() methods are just like the ones for the HomogeneousVolumeDensity and so aren't included here.

The task of the Density() method of the VolumeGridDensity is to use the samples to reconstruct the volume density function at the given point.

⟨*VolumeGridDensity Method Definitions*⟩ ≡
```
float VolumeGridDensity::Density(const Point &Pobj) const {
    if (!extent.Inside(Pobj)) return 0;
    ⟨Compute voxel coordinates and offsets for Pobj 594⟩
    ⟨Trilinearly interpolate density values to compute local density 594⟩
}
```

Given the eight sample values that surround a point in 3D, this method trilinearly interpolates them to compute the density function's value at the point. It starts by using BBox::Offset() to find the $[0, 1]^3$ offset of the point inside the bounding box and then scaling by (nx, ny, nz) to find the closest volume sample whose integer coordinates are all less than the sample location. It then uses the Manhattan distances along each axis, (dx, dy, and dz), as the interpolants. Note that the same conventions for discrete versus continuous texel coordinates are used here as were used in the ImageFilm and MIPMap (Section 7.1.7).

BBox 70
BBox::Inside() 72
BBox::Offset() 73
DensityRegion 591
HomogeneousVolumeDensity 589
ImageFilm 404
MIPMap 530
Point 63
Spectrum 263
Transform 76
VolumeGridDensity 593
VolumeGridDensity::
  extent 593

⟨*Compute voxel coordinates and offsets for* Pobj⟩ ≡ 593
```
Vector vox = extent.Offset(Pobj);
vox.x = vox.x * nx - .5f;
vox.y = vox.y * ny - .5f;
vox.z = vox.z * nz - .5f;
int vx = Floor2Int(vox.x), vy = Floor2Int(vox.y), vz = Floor2Int(vox.z);
float dx = vox.x - vx, dy = vox.y - vy, dz = vox.z - vz;
```

These distances can be used directly in a series of invocations of `Lerp()` to estimate the density at the sample point:

⟨*Trilinearly interpolate density values to compute local density*⟩ ≡ 593
```
float d00 = Lerp(dx, D(vx, vy, vz),     D(vx+1, vy, vz));
float d10 = Lerp(dx, D(vx, vy+1, vz),   D(vx+1, vy+1, vz));
float d01 = Lerp(dx, D(vx, vy, vz+1),   D(vx+1, vy, vz+1));
float d11 = Lerp(dx, D(vx, vy+1, vz+1), D(vx+1, vy+1, vz+1));
float d0 = Lerp(dy, d00, d10);
float d1 = Lerp(dy, d01, d11);
return Lerp(dz, d0, d1);
```

The `D()` utility method returns the density at the given sample position. Its only tasks are to handle out-of-bounds sample positions with clamping and to compute the appropriate array offset for the given sample. Unlike `MIPMaps`, clamping is almost always the desired solution for out-of-bounds coordinates here. Because all lookup points are inside the `VolumeGridDensity`'s bounding box, the only time we will have out-of-bounds coordinates is at the edges, either due to the offsets for linear interpolation or the continuous-to-discrete texel coordinate conversion. In both of these cases, clamping is the most sensible solution.

⟨*VolumeGridDensity Public Methods*⟩ +≡ 593
```
float D(int x, int y, int z) const {
    x = Clamp(x, 0, nx-1);
    y = Clamp(y, 0, ny-1);
    z = Clamp(z, 0, nz-1);
    return density[z*nx*ny + y*nx + x];
}
```

11.4.2 EXPONENTIAL DENSITY

Another useful density class is `ExponentialDensity`, which describes a density that varies as an exponential function of height h within a given 3D extent:

$$d(h) = ae^{-bh}.$$

The a and b values are parameters that control the overall density and how quickly it falls off as a function of height, respectively. This density function is a good model for the Earth's atmosphere as seen from the Earth's surface, where the atmosphere's curvature can generally be neglected. It can also be used to model low-lying fog at ground level. It is shown in Figure 11.16 and is defined in `volumes/exponential.h` and `volumes/exponential.cpp`.

BBox::Offset() 73
ExponentialDensity 595
Floor2Int() 1002
Lerp() 1000
MIPMap 530
Vector 57
VolumeGridDensity 593
VolumeGridDensity::D() 594
VolumeGridDensity::extent 593
VolumeGridDensity::nx 593
VolumeGridDensity::ny 593
VolumeGridDensity::nz 593

Figure 11.16: `ExponentialDensity` Used in a Scene with the Dragon Model. Note the reduction in density as a function of height. (Compare to Figure 11.14, for example.)

⟨*ExponentialDensity Declarations*⟩ ≡
```
class ExponentialDensity : public DensityRegion {
public:
    ⟨ExponentialDensity Public Methods 595⟩
private:
    ⟨ExponentialDensity Private Data 596⟩
};
```

The `ExponentialDensity` constructor initializes its member variables directly from its arguments. In addition to the volume scattering properties passed to the `DensityRegion` constructor, the volume's bound, and the a and b parameter values, this constructor takes a vector giving an "up" direction that orients the volume and is used to compute the height of points for the density computation. While the up direction is not strictly necessary (the world-to-object transformation is sufficient to orient the volume), specifying an explicit up vector can be conceptually easier for the user.

BBox 70
DensityRegion 591
ExponentialDensity 595
ExponentialDensity::extent 596
ExponentialDensity::upDir 596
Spectrum 263
Transform 76
Vector 57
Vector::Normalize() 63

⟨*ExponentialDensity Public Methods*⟩ ≡ 595
```
ExponentialDensity(const Spectrum &sa, const Spectrum &ss,
                   float gg, const Spectrum &emit, const BBox &e,
                   const Transform &v2w, float aa, float bb,
                   const Vector &up)
    : DensityRegion(sa, ss, gg, emit, v2w), extent(e), a(aa), b(bb) {
    upDir = Normalize(up);
}
```

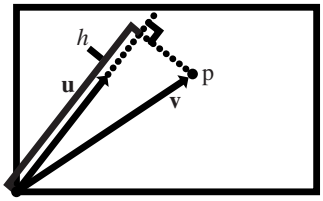

Figure 11.17: For the `ExponentialDensity`, it's necessary to find the perpendicular projection of the point p onto the "up" direction vector **u** and determine the distance h along **u** of the projection point. This distance is given by the dot product $(\mathbf{u} \cdot \mathbf{v})$, where **v** is the vector from the corner of the box (and the up direction's starting point) to p. This can be verified with basic properties of the dot product.

⟨*ExponentialDensity Private Data*⟩ ≡ 595
```
BBox extent;
float a, b;
Vector upDir;
```

The `ExponentialDensity::WorldBound()` and `ExponentialDensity::IntersectP()` methods are the same as their `HomogeneousVolumeDensity` counterparts and are not shown here.

The height of a given object space point p along the "up" direction axis can be found by projecting the vector from the lower corner of the bounding box to p onto the "up" direction vector (Figure 11.17). The distance h along the up direction to the point of p's perpendicular projection is given by the dot product of these two vectors. The principles behind this relationship are similar to those used when vectors are transformed to and from the BSDF coordinate system, for example.

⟨*ExponentialDensity Public Methods*⟩ +≡ 595
```
float Density(const Point &Pobj) const {
    if (!extent.Inside(Pobj)) return 0;
    float height = Dot(Pobj - extent.pMin, upDir);
    return a * expf(-b * height);
}
```

11.5 VOLUME AGGREGATES

Just as `Aggregate` implementations can hold sets of `Primitives`, the `AggregateVolume` holds one or more `VolumeRegions`. There are two main reasons to provide volume aggregates in `pbrt`. First, doing so simplifies the `Scene` and the implementation of `VolumeIntegrators`, since they can both be written to make calls to a single aggregate volume region, rather than looping over all of the regions in the scene. Second, `AggregateVolume` implementations have the potential to use 3D spatial data structures to improve efficiency by culling volumes that are far from a particular ray or lookup point.

Aggregate 192
AggregateVolume 597
BBox 70
ExponentialDensity 595
ExponentialDensity::a 596
ExponentialDensity::b 596
HomogeneousVolumeDensity 589
Point 63
Primitive 185
Scene 22
Vector 57
VolumeIntegrator 876
VolumeRegion 587

Here, we will just implement a simple AggregateVolume that stores an array of all the volumes in the scene and loops over them in each of its method implementations. This implementation is inefficient for scenes with many distinct VolumeRegions. Writing a more efficient implementation is left as an exercise at the end of the chapter.

⟨*Volume Scattering Declarations*⟩ +≡
```
class AggregateVolume : public VolumeRegion {
public:
    ⟨AggregateVolume Public Methods⟩
private:
    ⟨AggregateVolume Private Data 597⟩
};
```

The AggregateVolume constructor is simple. It copies the vector of VolumeRegions passed in and computes the bound that encloses them all.

⟨*Volume Scattering Definitions*⟩ +≡
```
AggregateVolume::AggregateVolume(const vector<VolumeRegion *> &r) {
    regions = r;
    for (uint32_t i = 0; i < regions.size(); ++i)
        bound = Union(bound, regions[i]->WorldBound());
}
```

⟨*AggregateVolume Private Data*⟩ ≡ 597
```
vector<VolumeRegion *> regions;
BBox bound;
```

As described earlier, the implementations of most of the various VolumeRegion interface methods loop over the individual regions. We will show AggregateVolume::sigma_a() as an example; the rest are similar and not shown here. Note that it forwards the call on to each contained VolumeRegion and adds the results. This approach works as desired because the individual VolumeRegion::sigma_a() methods return zero for points outside of their respective extents.

⟨*Volume Scattering Definitions*⟩ +≡
```
Spectrum AggregateVolume::sigma_a(const Point &p, const Vector &w,
                                  float time) const {
    Spectrum s(0.);
    for (uint32_t i = 0; i < regions.size(); ++i)
        s += regions[i]->sigma_a(p, w, time);
    return s;
}
```

AggregateVolume 597
AggregateVolume::regions 597
AggregateVolume::sigma_a() 597
BBox 70
Point 63
Spectrum 263
Vector 57
VolumeRegion 587
VolumeRegion::sigma_a() 588

The one method of this class that isn't completely trivial is IntersectP(). The parametric t range of the ray over all the volumes is equal to the extent from the minimum of all of the regions' t_{min} values to the maximum of all of the t_{max} values.

⟨*Volume Scattering Definitions*⟩ +≡
```
bool AggregateVolume::IntersectP(const Ray &ray,
                                 float *t0, float *t1) const {
    *t0 = INFINITY;
    *t1 = -INFINITY;
    for (uint32_t i = 0; i < regions.size(); ++i) {
        float tr0, tr1;
        if (regions[i]->IntersectP(ray, &tr0, &tr1)) {
            *t0 = min(*t0, tr0);
            *t1 = max(*t1, tr1);
        }
    }
    return (*t0 < *t1);
}
```

11.6 THE BSSRDF

The BSSRDF class plays a role similar to that of the BSDF in pbrt. While the BSDF describes scattering from a surface due to illumination at a single point, the BSSRDF describes the scattering properties for illumination that arrives at many points on the surface and undergoes subsurface scattering before exiting at the point being shaded. During the shading process, BSSRDF pointers are returned by the Primitive::GetBSSRDF() method if the object's material exhibits subsurface scattering.

⟨*BSSRDF Declarations*⟩ ≡
```
class BSSRDF {
public:
    ⟨BSSRDF Public Methods 598⟩
private:
    ⟨BSSRDF Private Data 599⟩
};
```

Unlike the BSDF class, which provides methods to evaluate the value of the BSDF given pairs of angles, the BSSRDF only describes the lower-level scattering properties of the participating medium. It's up to the integrator that computes the effect of subsurface scattering to use these properties in whichever algorithm it uses to compute the value of the BSSRDF for given pairs of incident and exitant locations and directions. (For example, the DipoleSubsurfaceIntegrator of Section 16.5 uses them to compute a BSSRDF approximation based on a closed-form solution for a semi-infinite slab.)

The properties that define the BSSRDF here are the index of refraction of the medium η, its absorption coefficient σ_a, and its *reduced scattering coefficient*, which is defined as $\sigma_s' = (1 - g)\sigma_s$, where g is the medium's anisotropy parameter and σ_s is the scattering coefficient. The reduced scattering coefficient is discussed further in Section 16.5.3.

⟨*BSSRDF Public Methods*⟩ ≡ 598
```
BSSRDF(const Spectrum &sa, const Spectrum &sps, float et)
    : e(et), sig_a(sa), sigp_s(sps) { }
```

AggregateVolume 597
AggregateVolume::regions 597
BSDF 478
BSSRDF 598
DipoleSubsurfaceIntegrator 887
INFINITY 1002
Primitive::GetBSSRDF() 187
Ray 66
Spectrum 263
VolumeRegion::IntersectP() 588

⟨*BSSRDF Private Data*⟩ ≡ 598
```
float e;
Spectrum sig_a, sigp_s;
```

The BSSRDF provides methods to get the values of the three properties. Note that the abstraction here implicitly treats these 3D properties of the medium as properties that only vary on the surface, since the `Material` returns a `BSSRDF`, and the `BSSRDF` doesn't represent any spatial variation of the scattering properties. Effectively, the entire medium is treated as if it were homogeneous, with the scattering properties computed at the point being shaded. This limitation is fine for homogeneous scattering media, but is inaccurate for highly inhomogeneous media.

⟨*BSSRDF Public Methods*⟩ +≡ 598
```
float eta() const { return e; }
Spectrum sigma_a() const { return sig_a; }
Spectrum sigma_prime_s() const { return sigp_s; }
```

11.6.1 SUBSURFACE SCATTERING MATERIALS

There are two `Materials` for translucent objects: `SubsurfaceMaterial`, which is defined in `materials/subsurface.h` and `materials/subsurface.cpp`, and `KdSubsurfaceMaterial`, defined in `materials/kdsubsurface.h` and `materials/kdsubsurface.cpp`. The only difference between these two materials is how the scattering properties of the medium are specified.

⟨*SubsurfaceMaterial Declarations*⟩ ≡
```
class SubsurfaceMaterial : public Material {
public:
    ⟨SubsurfaceMaterial Public Methods⟩
private:
    ⟨SubsurfaceMaterial Private Data 599⟩
};
```

`SubsurfaceMaterial` stores textures that allow the scattering properties to vary as a function of the position on the surface. Note that this isn't the same as scattering properties that vary as a function of 3D inside the scattering medium, but it can give a reasonable approximation to heterogeneous media in some cases.

In addition to the volumetric scattering properties, a specular reflection term `Kr` accounts for specular reflection from incident rays that don't enter the medium due to the different indices of refraction at the boundary. The `SubsurfaceMaterial::GetBSDF()` method adds a `SpecularReflection` component to the `BSDF` if `Kr` isn't black.

⟨*SubsurfaceMaterial Private Data*⟩ ≡ 599
```
float scale;
Reference<Texture<Spectrum> > Kr, sigma_a, sigma_prime_s;
Reference<Texture<float> > eta, bumpMap;
```

The `SubsurfaceMaterial::GetBSSRDF()` method uses the textures to compute the values of the scattering properties at the point. The absorption and reduced scattering coefficients are then scaled by the `scale` member variable, which provides an easy way to

BSDF 478
BSSRDF 598
KdSubsurfaceMaterial 600
Material 483
Reference 1011
Spectrum 263
SpecularReflection 440
SubsurfaceMaterial 599
Texture 519

change the units of the scattering properties. (Recall that they're expected to be specified in terms of reciprocal distance measured in meters, m^{-1}. If the scene isn't modeled in meters, the scattering properties can be adjusted accordingly.)

⟨*SubsurfaceMaterial Method Definitions*⟩ ≡
```
BSSRDF *SubsurfaceMaterial::GetBSSRDF(const DifferentialGeometry &dgGeom,
        const DifferentialGeometry &dgShading, MemoryArena &arena) const {
    float e = eta->Evaluate(dgShading);
    return BSDF_ALLOC(arena, BSSRDF)(scale * sigma_a->Evaluate(dgShading),
        scale * sigma_prime_s->Evaluate(dgShading), e);
}
```

Directly setting the absorption and reduced scattering coefficients to achieve a desired visual look is difficult. The parameters have a nonlinear and unintuitive effect on the result. The `KdSubsurfaceMaterial` allows the user to specify the subsurface scattering properties in terms of the diffuse reflectance of the surface and the mean free path—the average distance light travels in the medium before scattering. It then uses the `SubsurfaceFromDiffuse()` utility function, defined in Section 16.5, to compute the corresponding intrinsic scattering properties. Being able to specify translucent materials in this manner is particularly useful in that it makes it possible to use standard texture maps that might otherwise be used for diffuse reflection to define scattering properties (again with the caveat that varying properties on the surface don't properly correspond to varying properties in the medium).

We won't include the definition of `KdSubsurfaceMaterial` here since its implementation just evaluates `Textures` to compute the diffuse reflection and mean free path values, and calls `SubsurfaceFromDiffuse()` to compute the scattering properties needed by the `BSSRDF`.

Finally, `GetVolumeScatteringProperties()` is a utility function that has a small library of measured scattering data for translucent materials; it returns the corresponding scattering properties if it has an entry for the given name. (For a list of the valid names, see the implementation in `core/volume.cpp`.) The data provided by this function is from papers by Jensen et al. (2001b) and Narasimhan et al. (2006).

⟨*Volume Scattering Declarations*⟩ +≡
```
bool GetVolumeScatteringProperties(const string &name, Spectrum *sigma_a,
                                   Spectrum *sigma_prime_s);
```

BSSRDF 598
DifferentialGeometry 102
MemoryArena 1015
Spectrum 263
SubsurfaceFromDiffuse() 913
SubsurfaceMaterial::eta 599
SubsurfaceMaterial::scale 599
SubsurfaceMaterial::sigma_a 599
SubsurfaceMaterial::sigma_prime_s 599
Texture 519
Texture::Evaluate() 520

FURTHER READING

The books written by van de Hulst (1980) and Preisendorfer (1965, 1976) are excellent introductions to volume light transport. The seminal book by Chandrasekhar (1960) is another excellent resource, although it is mathematically challenging. See also the "Further Reading" section of Chapter 16 for more references to this topic.

The Henyey–Greenstein phase function was originally described by Henyey and Greenstein (1941). Detailed discussion of scattering and phase functions, along with derivations of phase functions that describe scattering from independent spheres, cylinders,

and other simple shapes, can be found in van de Hulst's book (1981). Extensive discussion of the Mie and Rayleigh scattering models is also available there. Hansen and Travis's survey article is also a good introduction to the variety of commonly used phase functions (Hansen and Travis 1974).

Just as procedural modeling of textures is an effective technique for shading surfaces, procedural modeling of volume densities can be used to describe realistic-looking volumetric objects like clouds and smoke. Perlin and Hoffert (1989) described early work in this area, and the book by Ebert et al. (2003) has a number of sections devoted to this topic, including further references. More recently, accurate physical simulation of the dynamics of smoke and fire has led to extremely realistic volume data sets, including the ones used in this chapter; see, for example, Fedkiw, Stam, and Jensen (2001).

In this chapter, we have ignored all issues related to sampling and antialiasing of volume density functions that are represented by samples in a 3D grid, although these issues should be considered, especially in the case of a volume that occupies just a few pixels on the screen. Furthermore, we have used a simple triangle filter to reconstruct densities at intermediate positions, which is suboptimal for the same reasons as the triangle filter is not a high-quality image reconstruction filter. Marschner and Lobb (1994) presented the theory and practice of sampling and reconstruction for 3D data sets, applying ideas similar to those in Chapter 7. See also the paper by Theußl, Hauser, and Gröller (2000) for a comparison of a variety of windowing functions for volume reconstruction with the sinc function and a discussion of how to derive optimal parameters for volume reconstruction filter functions.

Acquiring volumetric scattering properties of real-world objects is particularly difficult, requiring solving the inverse problem of determining the values that lead to the measured result. See Jensen et al. (2001b), Goesele et al. (2004), Narasimhan et al. (2006), and Peers et al. (2006) for recent work on acquiring scattering properties for subsurface scattering. Hawkins et al. (2005) have developed techniques to measure properties of media like smoke, acquiring measurements in real time. Another interesting approach to this problem was introduced by Frisvad et al. (2007), who developed methods to compute these properties from a lower-level characterization of the scattering properties of the medium.

EXERCISES

❶ 11.1 The optical thickness of a ray passing through an `ExponentialDensity` can be computed in closed form, so that the default `tau()` method based on Monte Carlo integration isn't needed. Derive this expression and add an implementation that computes its value to the `ExponentialDensity` class. Test your implementation to ensure that it computes the same results as the Monte Carlo approach. How much does this speed up `pbrt` for scenes that use an instance of the `ExponentialDensity` volume?

ExponentialDensity 595

❷ 11.2 Given a one-dimensional volume density that is an arbitrary function of height $f(h)$, the optical distance between any two three-dimensional points can be computed very efficiently if the integral $\int_0^{h'} f(h)\,\mathrm{d}h$ is precomputed and stored

in a table for a set of h' values (Perlin 1985b; Max 1986; Legakis 1998). Work through the mathematics to show the derivation for this approach and implement it in pbrt, either by modifying ExponentialDensity to use such a lookup table or by implementing a new VolumeRegion that takes an arbitrary function or a 1D table of density values. How do the efficiency and accuracy of this approach compare to using the default implementation of DensityRegion::tau()?

● 11.3 The VolumeGridDensity class uses a relatively large amount of memory for complex volume densities. Determine its memory requirements when used for the smoke images in this chapter and modify its implementation to reduce memory use. One approach is to detect regions of space with constant (or relatively constant) density values using an octree data structure and to only refine the octree in regions where the densities are changing. Another possibility is to use less memory to record each density value, for example, by computing the minimum and maximum densities and then using 8 or 16 bits per density value to interpolate between them. What sort of errors appear when either of these approaches is pushed too far?

● 11.4 Modify VolumeGridDensity to use an optimized version of the tau() method instead of relying on the generic Monte Carlo-based implementation of DensityRegion::tau(). Incorporate the available information about the distribution of densities to compute the integral using fewer samples in areas where the density is low. For inspiration, see, for example, Levoy (1988), which describes using a binary volume octree to speed up traversal of empty regions, and Danskin and Hanrahan (1992), which describes various techniques based on a 3D pyramid of volume data to use lower-precision computations in regions that have a low contribution to the final result.

● 11.5 The AggregateVolume will have poor performance for a scene with more than a handful of VolumeRegions. For example, time will be wasted determining the values of σ_t and σ_s in areas where the point is outside most of the volumes. Write a better volume aggregate based on a 3D data structure like a grid or an octree. Verify that it returns the same results as the AggregateVolume and measure how much faster pbrt is when it is used instead of AggregateVolume. (Don't forget to modify the RenderOptions::MakeScene() method to create an instance of your new aggregate instead of an AggregateVolume.)

● 11.6 Implement the atmospheric scattering model described by Preetham, Shirley, and Smits (1999) in pbrt. Use it to render images of the ecosystem scene and other outdoor scenes.

AggregateVolume 597
DensityRegion::tau() 733
ExponentialDensity 595
RenderOptions::
MakeScene() 1072
VolumeGridDensity 593
VolumeRegion 587