CHAPTER SIXTEEN

★ 16 LIGHT TRANSPORT II: VOLUME RENDERING

Just as `SurfaceIntegrators` are the meeting point of scene geometry, materials, and lights, applying sophisticated algorithms to solve the light transport equation and determine the distribution of radiance in the scene, `VolumeIntegrators` are responsible for incorporating the effect of participating media (as described by `VolumeRegions`) into this process and determining how it affects the distribution of radiance. This chapter briefly introduces the equation of transfer, which describes how participating media change radiance along rays, and then describes the `VolumeIntegrator` interface as well as a few simple `VolumeIntegrator` implementations. Section 16.5 then describes the implementation of a surface integrator that accounts for the effect of *subsurface scattering*, where incident light travels some distance inside a surface before exiting. Although the approach is implemented as a `SurfaceIntegrator`, it is included in this chapter since its implementation is based on an approximate solution to light transport through participating media.

Many of the general approaches to light transport algorithms for surfaces described in the previous chapter can be extended to handle participating media as well. The "Further Reading" section and this chapter's exercises discuss these connections further.

16.1 THE EQUATION OF TRANSFER

The equation of transfer is the fundamental equation that governs the behavior of light in a medium that absorbs, emits, and scatters radiation (Chandrasekhar 1960). It accounts for all of the volume scattering processes described in Chapter 11—absorption, emission, and in- and out-scattering—to give an equation that describes the distribution of radiance in an environment. The light transport equation is in fact a special case of the equation of transfer, simplified by the lack of participating media and specialized for scattering from surfaces (Arvo 1993).

In its most basic form, the equation of transfer is an integro-differential equation that describes how the radiance along a beam changes at a point in space. It can be transformed into a pure integral equation that describes the effect of participating media from the infinite number of points along a line. It can be derived in a straightforward manner by subtracting the effects of the scattering processes that reduce energy along a beam (absorption and out-scattering) from the processes that increase energy along it (emission and in-scattering). Recall the source term from Section 11.1.4: it gives the change in radiance at a point p in a particular direction ω due to emission and in-scattered light from other points in the medium:

$$S(\mathrm{p}, \omega) = L_{\mathrm{ve}}(\mathrm{p}, \omega) + \sigma_{\mathrm{s}}(\mathrm{p}, \omega) \int_{\mathbb{S}^2} p(\mathrm{p}, -\omega' \to \omega) L_{\mathrm{i}}(\mathrm{p}, \omega') \, \mathrm{d}\omega'.$$

The source term accounts for all of the processes that add radiance to a ray.

The attenuation coefficient, $\sigma_{\mathrm{t}}(\mathrm{p}, \omega)$, accounts for all processes that reduce radiance at a point: absorption and out-scattering. The differential equation that describes its effect is

$$\mathrm{d}L_{\mathrm{o}}(\mathrm{p}, \omega) = -\sigma_{\mathrm{t}}(\mathrm{p}, \omega) L_{\mathrm{i}}(\mathrm{p}, -\omega) \mathrm{d}t.$$

The overall differential change in radiance at a point p' along a ray is found by adding these two effects together to get the integro-differential form of the equation of transfer:[1]

$$\frac{\partial}{\partial t} L_{\mathrm{o}}(\mathrm{p}, \omega) = -\sigma_{\mathrm{t}}(\mathrm{p}, \omega) L_{\mathrm{i}}(\mathrm{p}, -\omega) + S(\mathrm{p}, \omega).$$

With suitable boundary conditions, this equation can be transformed to a purely integral equation. For example, if we assume that there are no surfaces in the scene so that the rays are never blocked and have an infinite length, the integral equation of transfer is

$$L_{\mathrm{i}}(\mathrm{p}, \omega) = \int_0^\infty T_r(\mathrm{p}' \to \mathrm{p}) S(\mathrm{p}', -\omega) \, \mathrm{d}t,$$

where $\mathrm{p}' = \mathrm{p} + t\omega$ (Figure 16.1). The meaning of this equation is reasonably intuitive: it just says that the radiance arriving at a point from a given direction is contributed to by the added radiance along all points along the ray from the point. The amount of added radiance at each point along the ray that reaches the ray's origin is reduced by the total beam transmittance from the ray's origin to the point.

More generally, if there are reflecting and/or emitting surfaces in the scene, rays don't necessarily have infinite length and the surface that a ray hits affects its radiance, adding outgoing radiance from the surface at the point and preventing radiance from points along the ray beyond the intersection point from contributing to radiance at the ray's origin. If a ray (p, ω) intersects a surface at some point p_0 a distance t along the ray, then the integral equation of transfer is

$$L_{\mathrm{i}}(\mathrm{p}, \omega) = T_r(\mathrm{p}_0 \to \mathrm{p}) L_{\mathrm{o}}(\mathrm{p}_0, -\omega) + \int_0^t T_r(\mathrm{p}' \to \mathrm{p}) S(\mathrm{p}', -\omega) \mathrm{d}t', \qquad [16.1]$$

1 It is an integro-differential equation due to the integral over the sphere in the source term.

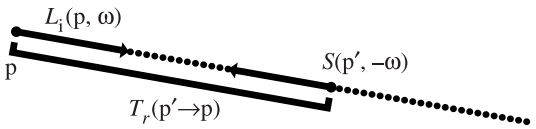


Figure 16.1: The equation of transfer gives the incident radiance at point $L_i(p, \omega)$ accounting for the effect of participating media. At each point along the ray, the source term $S(p', \omega)$ gives the differential radiance added at the point due to scattering and emission. This radiance is then attenuated by the beam transmittance $T_r(p' \to p)$ from the point p' to the ray's origin.

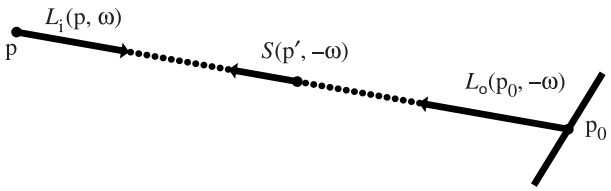


Figure 16.2: For a finite ray that intersects a surface, the incident radiance, $L_i(p, \omega)$, is equal to the outgoing radiance from the surface, $L_o(p_0, -\omega)$, times the beam transmittance to the surface plus the added radiance from all points along the ray from p to p_0.

where $p_0 = p + t\omega$ is the point on the surface and $p' = p + t'\omega$ are points along the ray (Figure 16.2).

This equation describes the two effects that contribute to radiance along the ray. First, reflected radiance back along the ray from the surface is given by the L_o term, which gives the emitted and reflected radiance from the surface. This radiance may be attenuated by the participating media; the beam transmittance from the ray origin to the point p_0 accounts for this. The second term accounts for the added radiance along the ray due to volume scattering and emission, but only up to the point where the ray intersects the surface; points beyond that one don't affect the radiance along the ray.

In the interest of brevity, we will refrain from further generalization of the equation of transfer here. However, just as the light transport equation could be written in a more general form as a sum over paths of various numbers of vertices and just as an importance function could be introduced to turn it into the measurement equation, the equation of transfer can be generalized in a similar manner. Likewise, we will only present a few simple `VolumeIntegrators` in most of this chapter, although the general types of algorithms used for the surface integrators in the previous chapter, such as path tracing, bidirectional path tracing, photon mapping, and so on, can be applied to volume integration as well.

VolumeIntegrator 876

16.2 VOLUME INTEGRATOR INTERFACE

The `VolumeIntegrator` interface inherits from `Integrator`, picking up the `Preprocess()`, and `RequestSamples()` methods declared there. These two methods are used by volume integrators in the same way as by surface integrators.

`VolumeIntegrator` adds a `Li()` method that is similar to the surface integrator version in that it returns the radiance along the given ray; volume integrators should assume that if the given ray does intersect a surface in the scene, then its `Ray::maxt` value will have been set to be at the intersection point. As such, the volume integrator should only compute the effect of volume scattering from the parametric range `[mint, maxt]` along the ray. The `VolumeIntegrator`'s version of `Li()` doesn't include the `Intersection` parameter that `SurfaceIntegrator` has, and it adds an output `Spectrum` parameter to return the transmittance from the start of the ray to the end of the ray. This returned transmittance value is used to compute the attenuation of outgoing radiance from the intersected surface. Like `SurfaceIntegrator::Li()`, the `Sample` pointer may be `NULL`, in which case the implementation should use the provided `RNG` for any random sampling it performs.

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

The `VolumeIntegrator` interface adds an additional method that implementations must provide, `Transmittance()`, which is responsible for computing the beam transmittance along the given ray from `Ray::mint` to `Ray::maxt`. This method is used to compute attenuation along shadow rays traced for direct lighting, for example.

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

With this background, the `SamplerRenderer::Li()` method can be fully understood. It is a direct implementation of Equation (16.1). The surface integrator computes outgoing radiance L_o at the ray's intersection point, ignoring attenuation back to the ray origin. The volume integrator's `Li()` method gives the radiance along the ray due to participating media and also returns the beam transmittance T_r to the point on the surface. The sum of $L_o T_r$ and the additional radiance from participating media gives the total radiance arriving at the ray origin.

16.3 EMISSION-ONLY INTEGRATOR

The simplest possible volume integrator (other than one that ignored participating media completely) simplifies the source term by ignoring in-scattering and only accounting for emission and attenuation. Because in-scattering is ignored, the integral over the sphere in the source term at each point along the ray disappears, and the resulting sim-

Integrator 740
Intersection 186
MemoryArena 1015
Ray::maxt 67
Ray::mint 67
RayDifferential 69
Renderer 24
RNG 1003
Sample 343
SamplerRenderer::Li() 34
Scene 22
Spectrum 263
SurfaceIntegrator 740
SurfaceIntegrator::Li() 741
VolumeIntegrator 876

plified equation of transfer is

$$L_i(p, \omega) = T_r(p_0 \to p) L_o(p_0, -\omega) + \int_0^t T_r(p' \to p) L_{ve}(p', -\omega)\, dt. \quad [16.2]$$

The `EmissionIntegrator`, which is defined in the files `integrators/emission.h` and `integrators/emission.cpp`, uses Monte Carlo integration to solve this equation. Figure 16.3(a) shows the ecosystem scene without any participating media; Figure 16.3(b), with fog rendered with the `EmissionIntegrator`.

⟨*EmissionIntegrator Declarations*⟩ ≡
```
class EmissionIntegrator : public VolumeIntegrator {
public:
    ⟨EmissionIntegrator Public Methods 877⟩
private:
    ⟨EmissionIntegrator Private Data 877⟩
};
```

The `EmissionIntegrator`'s `Transmittance()` and `Li()` methods both have to evaluate one-dimensional integrals along points t' along a ray. Rather than using a fixed number of samples for each estimate, the implementation here bases the number of samples on the distance the ray travels in the volume—the longer the distance, the more samples are taken. This approach is worthwhile for naturally intuitive reasons—the longer the ray's extent in the medium, the more accuracy is desirable and the more samples are likely to be needed to capture greater variation in optical properties along the ray. The number of samples taken is determined indirectly by a user-supplied parameter giving a step size between samples. The ray is divided into segments of the given length, and a single sample is taken in each one.

⟨*EmissionIntegrator Public Methods*⟩ ≡ 877
```
EmissionIntegrator(float ss) { stepSize = ss; }
```

⟨*EmissionIntegrator Private Data*⟩ ≡ 877
```
float stepSize;
```

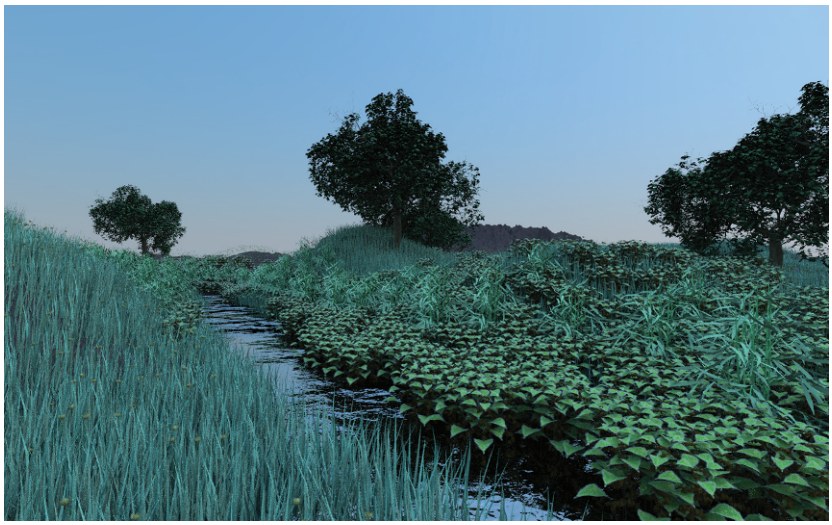
The `Transmittance()` and `Li()` methods each only need a single 1D sample value to evaluate their respective integrals:

⟨*EmissionIntegrator Method Definitions*⟩ ≡
```
void EmissionIntegrator::RequestSamples(Sampler *sampler, Sample *sample,
                                        const Scene *scene) {
    tauSampleOffset = sample->Add1D(1);
    scatterSampleOffset = sample->Add1D(1);
}
```

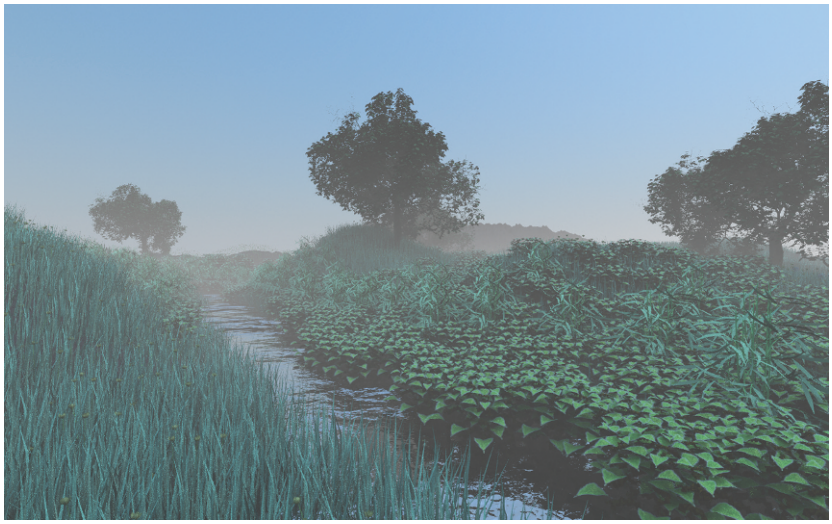
⟨*EmissionIntegrator Private Data*⟩ +≡ 877
```
int tauSampleOffset, scatterSampleOffset;
```

The `Transmittance()` method is reasonably straightforward. The `VolumeRegion`'s `tau()` method takes care of computing the optical thickness τ from the ray's starting point to its ending point. The only work for the integrator here is to choose a step size (in case `tau()` does Monte Carlo integration, as the implementation in Section 14.7 does), pass a

EmissionIntegrator 877
EmissionIntegrator::scatterSampleOffset 877
EmissionIntegrator::stepSize 877
EmissionIntegrator::tauSampleOffset 877
Sample 343
Sample::Add1D() 344
Sampler 340
Scene 22
VolumeIntegrator 876
VolumeRegion 587



(a)



(b)

Figure 16.3: Ecosystem scene rendered (a) without any participating media and (b) with `ExponentialDensity`-based fog rendered with the `EmissionIntegrator`. Participating media makes the image look substantially more realistic by reducing the contrast of faraway objects.

single sample value to that method, and return $e^{-\tau}$. If the `tau()` method can compute τ analytically, it will ignore these additional values.

`EmissionIntegrator` 877

`ExponentialDensity` 595

This method takes advantage of the fact that the `Sample` value is only non-`NULL` for camera rays to increase the step size for shadow and indirect rays, thus reducing computa-

tional demands (and accuracy). The reduction in accuracy for those rays generally isn't noticeable.

⟨*EmissionIntegrator Method Definitions*⟩ +≡

```
Spectrum EmissionIntegrator::Transmittance(const Scene *scene,
        const Renderer *renderer, const RayDifferential &ray,
        const Sample *sample, RNG &rng, MemoryArena &arena) const {
    if (!scene->volumeRegion) return Spectrum(1.f);
    float step, offset;
    if (sample) {
        step = stepSize;
        offset = sample->oneD[tauSampleOffset][0];
    }
    else {
        step = 4.f * stepSize;
        offset = rng.RandomFloat();
    }
    Spectrum tau = scene->volumeRegion->tau(ray, step, offset);
    return Exp(-tau);
}
```

The `Li()` method is responsible for evaluating the second term of the sum in Equation (16.2). If the ray enters the volume at $t = t_0$ along the ray, then no attenuation or emission happens from the start of the ray up to t_0, and the `Li()` method can instead consider the integral from t_0 to t_1, where t_1 is the minimum of the parametric offset where the ray exits the volume and the offset where it intersects a surface (Figure 16.4). An estimate of the value of this integral,

$$\int_{t_0}^{t_1} T_r(\mathrm{p}' \to \mathrm{p}) L_{\mathrm{ve}}(\mathrm{p}', -\omega)\, \mathrm{d}t',$$

EmissionIntegrator 877
EmissionIntegrator::stepSize 877
EmissionIntegrator::tauSampleOffset 877
MemoryArena 1015
RayDifferential 69
Renderer 24
RNG 1003
RNG::RandomFloat() 1003
Sample 343
Sample::oneD 344
Scene 22
Scene::volumeRegion 23
Spectrum 263
Spectrum::Exp() 265
VolumeRegion::tau() 588

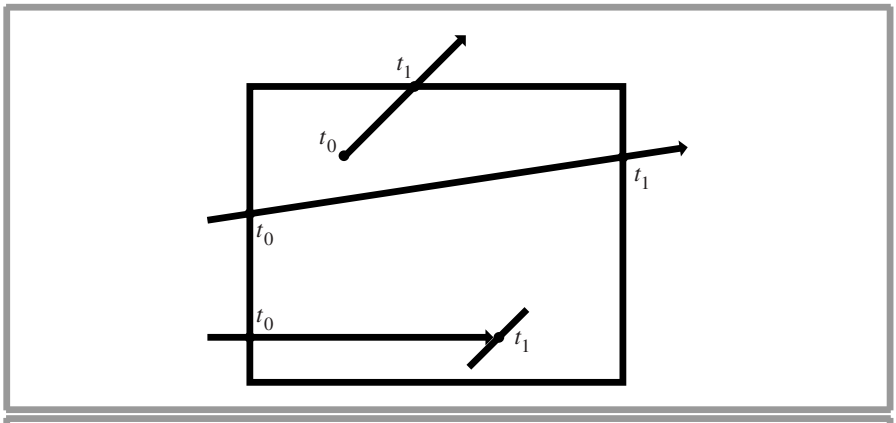


Figure 16.4: The starting point of the ray marching process, t_0, is the maximum of the ray's origin or the point where the ray enters the participating medium. Similarly, the end, t_1, is the minimum of the point where the ray exits the medium and the point where it intersects a surface, if any.

can be found by uniformly selecting random points p_i along the ray between t_0 and t_1 and evaluating the estimator

$$\frac{1}{N}\sum_i \frac{T_r(\mathrm{p}_i \to \mathrm{p})L_{\mathrm{ve}}(\mathrm{p}_i, -\omega)}{p(\mathrm{p}_i)} = \frac{t_1 - t_0}{N}\sum_i T_r(\mathrm{p}_i \to \mathrm{p})L_{\mathrm{ve}}(\mathrm{p}_i, -\omega)$$

since for uniformly sampled points $p(\mathrm{p}_i) = 1/(t_1 - t_0)$. The L_{ve} term in the estimator can be evaluated directly with the corresponding `volumeRegion` method, and the optical thickness τ to evaluate T_r can be evaluated either directly (for a homogeneous or exponential atmosphere), or via Monte Carlo integration as described in Section 14.7.

In order to do this computation, the implementation of the `Li()` method thus starts out by finding the t range for the integral and initializing `t0` and `t1` accordingly:

⟨*EmissionIntegrator Method Definitions*⟩ +≡
```
Spectrum EmissionIntegrator::Li(const Scene *scene,
        const Renderer *renderer, const RayDifferential &ray,
        const Sample *sample, RNG &rng, Spectrum *T,
        MemoryArena &arena) const {
    VolumeRegion *vr = scene->volumeRegion;
    float t0, t1;
    if (!vr || !vr->IntersectP(ray, &t0, &t1)) {
        *T = Spectrum(1.f);
        return 0.f;
    }
    ⟨Do emission-only volume integration in vr 880⟩
}
```

Two additional techniques are used in the implementation here. First, just as the `VolumeRegion::tau()` method in Section 14.7 used a uniform step size between sample points, this implementation also steps uniformly for similar reasons. Second, the beam transmittance values T_r can be evaluated efficiently if the points p_i are sorted from the one closest to the ray origin p to the one farthest away. Then, the multiplicative property of T_r can be used to incrementally compute T_r from its value from the previous point:

$$T_r(\mathrm{p}_i \to \mathrm{p}) = T_r(\mathrm{p}_i \to \mathrm{p}_{i-1})T_r(\mathrm{p}_{i-1} \to \mathrm{p}).$$

Because $T_r(\mathrm{p}_i \to \mathrm{p}_{i-1})$ covers a shorter distance than $T_r(\mathrm{p}_i \to \mathrm{p})$, fewer samples can be used to estimate its value if it is evaluated with Monte Carlo. Both of these ideas are illustrated in Figure 16.5.

⟨*Do emission-only volume integration in* vr⟩ ≡ 880
```
Spectrum Lv(0.);
⟨Prepare for volume integration stepping 881⟩
for (int i = 0; i < nSamples; ++i, t0 += step) {
    ⟨Advance to sample at t0 and update T 881⟩
    ⟨Compute emission-only source term at p 882⟩
}
*T = Tr;
return Lv * step;
```

EmissionIntegrator 877
MemoryArena 1015
RayDifferential 69
Renderer 24
RNG 1003
Sample 343
Scene 22
Scene::volumeRegion 23
Spectrum 263
VolumeRegion 587
VolumeRegion::IntersectP() 588
VolumeRegion::tau() 588

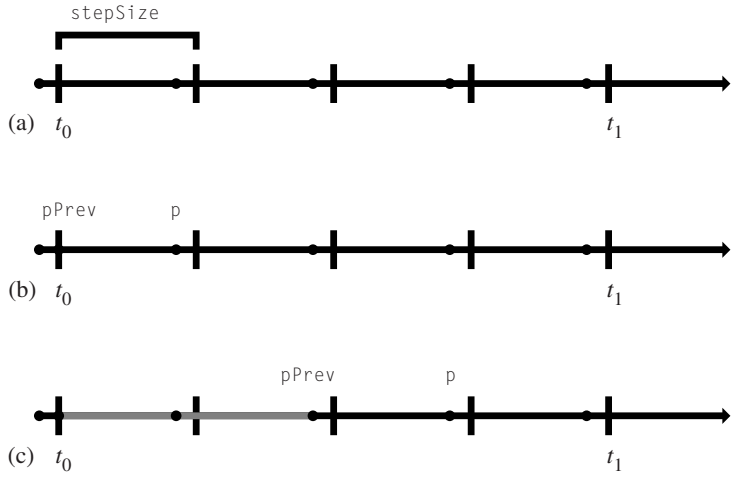


Figure 16.5: Volume Integration Ray Sampling. (a) The ray's extent from t_0 to t_1 is subdivided into a number of segments based on the user-supplied `stepSize` parameter. A single sample is taken in each segment, where the first sample is placed randomly in the first segment and all additional samples are offset by equal-sized steps. (b) Ray marching tracks the previous point to which transmittance was computed, `pPrev`, as well as the current point, `p`. Initially, `pPrev` is the point where the ray enters the volume. (c) At each subsequent step, beam transmittance is computed as the product of transmittance to `pPrev` and the additional transmittance from `pPrev` to `p`.

⟨*Prepare for volume integration stepping*⟩ ≡ 880
```
int nSamples = Ceil2Int((t1-t0) / stepSize);
float step = (t1 - t0) / nSamples;
Spectrum Tr(1.f);
Point p = ray(t0), pPrev;
Vector w = -ray.d;
t0 += sample->oneD[scatterSampleOffset][0] * step;
```

Ceil2Int() 1002
EmissionIntegrator::scatterSampleOffset 877
EmissionIntegrator::stepSize 877
Point 63
Ray 66
Ray::depth 67
Ray::time 67
RNG::RandomFloat() 1003
Sample::oneD 344
Spectrum 263
Spectrum::Exp() 265
Vector 57
VolumeRegion::tau() 588

To find the overall transmittance at the current point, it's only necessary to find the transmittance from the previous point to the current point and multiply it by the transmittance from the ray origin to the previous point, as described earlier.

⟨*Advance to sample at* `t0` *and update* `T`⟩ ≡ 880
```
pPrev = p;
p = ray(t0);
Ray tauRay(pPrev, p - pPrev, 0.f, 1.f, ray.time, ray.depth);
Spectrum stepTau = vr->tau(tauRay,
                           .5f * stepSize, rng.RandomFloat());
Tr *= Exp(-stepTau);
⟨Possibly terminate ray marching if transmittance is small 882⟩
```

In a thick medium, the transmittance may become very low after the ray has passed a sufficient distance through it. To reduce the time spent computing source term values

that are likely to have very little contribution to the radiance at the ray's origin, ray stepping is randomly terminated with Russian roulette when transmittance is sufficiently small.

⟨*Possibly terminate ray marching if transmittance is small*⟩ ≡ 881
```
if (Tr.y() < 1e-3) {
    const float continueProb = .5f;
    if (rng.RandomFloat() > continueProb) break;
    Tr /= continueProb;
}
```

Having done all this work, computing the source term at the point is easy:

⟨*Compute emission-only source term at* p⟩ ≡ 880
```
Lv += Tr * vr->Lve(p, w, ray.time);
```

16.4 SINGLE SCATTERING INTEGRATOR

In addition to accounting for the emission at each point along the ray, the `SingleScatteringIntegrator` also considers the incident radiance due to direct illumination but ignores incident radiance due to multiple scattering. Thus, its `Li()` method estimates the integral

$$\int_0^t T_r(\mathrm{p}' \to \mathrm{p}) \left(L_{\mathrm{ve}}(\mathrm{p}', -\omega) + \sigma_{\mathrm{s}}(\mathrm{p}', \omega) \int_{\mathbb{S}^2} p(\mathrm{p}', -\omega' \to -\omega) L_{\mathrm{d}}(\mathrm{p}', \omega') \mathrm{d}\omega' \right) \mathrm{d}t',$$

where L_{d} only includes radiance from direct lighting. This radiance may be blocked by geometry in the scene and may itself be attenuated by participating media between the light and the point p' along the ray (Figure 16.6). Although accounting for this effect can be much more computationally expensive than just ignoring it as the `EmissionIntegrator` does, it can give striking "beams of light" effects, as seen in Figures 11.1 and 16.7.

Almost all of the implementation of the `SingleScatteringIntegrator` parallels the `EmissionIntegrator`, so only the fragments in the parts that differ are included here. This integrator is defined in the files `integrators/single.h` and `integrators/single.cpp`.

⟨*SingleScatteringIntegrator Declarations*⟩ ≡
```
class SingleScatteringIntegrator : public VolumeIntegrator {
public:
    ⟨SingleScatteringIntegrator Public Methods⟩
private:
    ⟨SingleScatteringIntegrator Private Data⟩
};
```

This integrator's `Li()` method uses the same general ray-marching approach to evaluate the equation of transfer as the `EmissionIntegrator`. One difference is that this one computes sample values for light source sampling before it enters the `for` loop over sample positions. Because it isn't known how many samples will be necessary until `Li()` is called

EmissionIntegrator 877
RNG::RandomFloat() 1003
SingleScatteringIntegrator 882
Spectrum::y() 273
VolumeIntegrator 876
VolumeRegion::Lve() 588

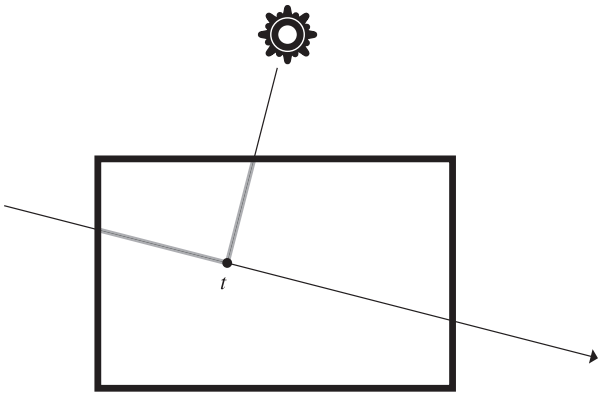


Figure 16.6: When the direct lighting contribution is evaluated at some point t along a ray passing through participating media, it's necessary to compute the attenuation of the radiance from the light passing through the volume to the scattering point as well as the attenuation from that point back to the ray origin.

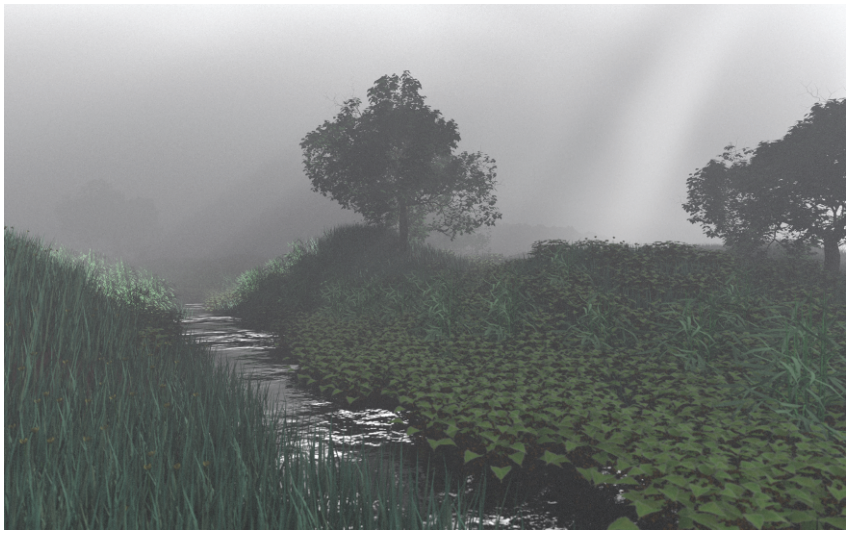


Figure 16.7: When the single scattering volume integrator is used with the ecosystem scene, beams of light are visible, giving a striking visual effect.

(since this number depends on the length of the ray segment over which integration is being done), it's not possible to have the `Sampler` generate samples and pass them into `Li()` via the `Sample`. Therefore, the samples are generated here. Two one-dimensional patterns are used for selecting which light to sample and which light component to sample, and a two-dimensional pattern is used for selecting points on area light sources.

⟨*Compute sample patterns for single scattering samples*⟩ ≡
```
float *lightNum = arena.Alloc<float>(nSamples);
LDShuffleScrambled1D(1, nSamples, lightNum, rng);
float *lightComp = arena.Alloc<float>(nSamples);
LDShuffleScrambled1D(1, nSamples, lightComp, rng);
float *lightPos = arena.Alloc<float>(2*nSamples);
LDShuffleScrambled2D(1, nSamples, lightPos, rng);
uint32_t sampOffset = 0;
```

The other difference from the `EmissionIntegrator` is how the source term is evaluated. At each sample point along the ray, the following fragment computes the single scattering approximation of the source term at the point `p`. It serves the same role as the earlier ⟨*Compute emission-only source term at* `p`⟩ fragment. After including volume emission in the same way that the `EmissionIntegrator` does, it finds the value of σ_s at the point, selects a light to sample, and computes its contribution to scattering at the point. Because the source term is generally evaluated at many points along the ray, only a single light is sampled at each one, and its contribution is scaled by the number of lights, similar to the direct lighting integrator's "sample one light" strategy.

⟨*Compute single-scattering source term at* `p`⟩ ≡
```
Lv += Tr * vr->Lve(p, w, ray.time);
Spectrum ss = vr->sigma_s(p, w, ray.time);
if (!ss.IsBlack() && scene->lights.size() > 0) {
    int nLights = scene->lights.size();
    int ln = min(Floor2Int(lightNum[sampOffset] * nLights),
                 nLights-1);
    Light *light = scene->lights[ln];
    ⟨Add contribution of light due to scattering at p 884⟩
}
++sampOffset;
```

Computing the estimate of the direct lighting contribution at a point p involves estimating the integral

$$\sigma_s(\mathrm{p}, \omega) \int_{S^2} p(\mathrm{p}, -\omega' \rightarrow -\omega) L_d(\mathrm{p}, \omega')\, d\omega'.$$

Rather than sampling both the phase function and the light source and applying multiple importance sampling, the implementation here always lets the light choose a sample position on the light source and then computes the estimator directly. For media that aren't extremely anisotropic, this approach works well.

⟨*Add contribution of* `light` *due to scattering at* `p`⟩ ≡ 884
```
float pdf;
VisibilityTester vis;
Vector wo;
LightSample ls(lightComp[sampOffset], lightPos[2*sampOffset],
               lightPos[2*sampOffset+1]);
Spectrum L = light->Sample_L(p, 0.f, ls, ray.time, &wo, &pdf, &vis);
```

Floor2Int() 1002
LDShuffleScrambled1D() 377
LDShuffleScrambled2D() 377
Light 606
Light::Sample_L() 608
LightSample 710
MemoryArena::Alloc() 1016
Scene::lights 23
Spectrum 263
Spectrum::IsBlack() 265
Vector 57
VisibilityTester 608
VisibilityTester::
  Transmittance() 609
VisibilityTester::
  Unoccluded() 609
VolumeRegion::Lve() 588
VolumeRegion::p() 588
VolumeRegion::sigma_s() 588

```
if (!L.IsBlack() && pdf > 0.f && vis.Unoccluded(scene)) {
    Spectrum Ld = L * vis.Transmittance(scene, renderer, NULL, rng, arena);
    Lv += Tr * ss * vr->p(p, w, -wo, ray.time) * Ld * float(nLights) /
            pdf;
}
```

16.5 SUBSURFACE SCATTERING

The bidirectional scattering-surface reflectance distribution function (BSSRDF) was introduced in Section 5.6.2; it gives exitant differential radiance at a point on a surface $\mathrm{p_o}$ given incident differential irradiance at another point $\mathrm{p_i}$: $S(\mathrm{p_o}, \omega_o, \mathrm{p_i}, \omega_i)$. Accurately rendering translucent surfaces with subsurface scattering requires integrating over both area—points on the surface of the object being rendered—and incident direction, evaluating the BSSRDF and computing reflection with the subsurface scattering equation

$$L_o(\mathrm{p_o}, \omega_o) = \int_A \int_{\mathcal{H}^2(\mathbf{n})} S(\mathrm{p_o}, \omega_o, \mathrm{p_i}, \omega_i)\, L_i(\mathrm{p_i}, \omega_i) |\cos \theta_i|\, \mathrm{d}\omega_i\, \mathrm{d}A.$$

Figure 16.8 suggests the complexity of evaluating the integral. To compute the standard Monte Carlo estimate of this equation given a point at which to compute outgoing radiance, we need a technique to sample points $\mathrm{p_i}$ on the surface and to compute the incident radiance at these points, as well as an efficient way to compute the specific value of the BSSRDF $S(\mathrm{p_o}, \omega_o, \mathrm{p_i}, \omega_i)$ for each sampled point $\mathrm{p_i}$ and incident direction.

For translucent materials, the scattering properties inside the material are described by the volume scattering mechanisms introduced in Section 11.1. The rendering problem is thus to take an irregularly shaped object where the boundary defines the extent of a

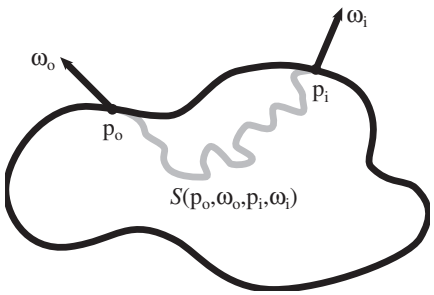


Figure 16.8: Computing Subsurface Reflection. When a surface is translucent, in order to compute outgoing radiance from a point $\mathrm{p_o}$ in direction ω_o, it's necessary to sample a number of points $\mathrm{p_i}$ and compute incident radiance at them. Then, the BSSRDF, $S(\mathrm{p_o}, \omega_o, \mathrm{p_i}, \omega_i)$, must be evaluated in order to determine the effect of incident radiance at one point on outgoing radiance at the point of interest. The BSSRDF can be difficult to evaluate efficiently, since it represents all scattering within the volume for light that enters at one point and exits at the other.

medium with these scattering properties, and to render images of the object accounting for the light transport in the medium. Thus, the BSSRDF S notationally represents the end effect of these scattering processes between a given pair of points and directions on the boundary.

A multiple scattering volume integrator can be used to evaluate the BSSRDF: given a pair of points on the surface and a pair of directions, the integrator can be used to compute the fraction of incident light from direction ω_i at the point p_i that exits the object at the point p_o in direction ω_o by following light-carrying paths through the multiple scattering events in the medium. Beyond standard path-tracing or bidirectional path-tracing techniques, many existing light transport algorithms are applicable to this task; for example, Jensen and Christensen (1998) applied volume photon maps.

However, many translucent objects are characterized by having very high albedos, which are not efficiently handled by classic approaches. For example, Jensen et al. (2001b) measured the scattering properties of skim milk and found an albedo of 0.9987. When essentially all of the light is scattered at each interaction in the medium and almost none of it is absorbed, light easily travels far from where it first enters the medium. Hundreds or even thousands of scattering events must be considered to compute an accurate result; given the high albedo of milk, after 100 scattering events, 87.5% of the incident light is still carried by a path, 51% after 500 scattering events, and still 26% after 1000.

Efficiently rendering these kinds of interesting translucent scattering media therefore requires a different approach to light transport than we've used so far. In this section, we'll present the implementation of an efficient method for rendering the effect of subsurface scattering in high-albedo materials. Figure 16.9 shows an example of the dragon model rendered with this integrator. Note that although light is arriving from above and slightly behind it, the effect of light traveling inside the medium and exiting far from where it entered is well modeled.

The method we'll implement in this section is based on two main ideas. First, the distribution of light in the translucent medium is modeled with the *diffusion approximation*, which describes the equlibrium distribution of illumination in highly scattering optically thick participating media. Second, a solution to the diffusion equation is found by composing closed-form solutions of it for point light sources; a *dipole* of two light sources is used to approximate the overall scattering.

This approach makes a number of simplifications—it assumes heterogeneous scattering properties throughout the medium, and it implicitly assumes that the medium is *semi-infinite* (it extends infinitely beneath a planar surface). In practice, however, it can still accurately render many interesting surfaces. The "Further Reading" section and exercises at the end of this chapter have references to improvements to this approach that generalize it to handle more general settings more accurately.

Our implementation of subsurface scattering has three main components, each of which will be described in its own section to follow:

1. A large number of random points on the surfaces of the translucent objects in the scene are sampled and incident irradiance is computed at these points.
2. A hierarchical representation of these points is created, progressively clustering nearby groups of them summing their irradiance values.

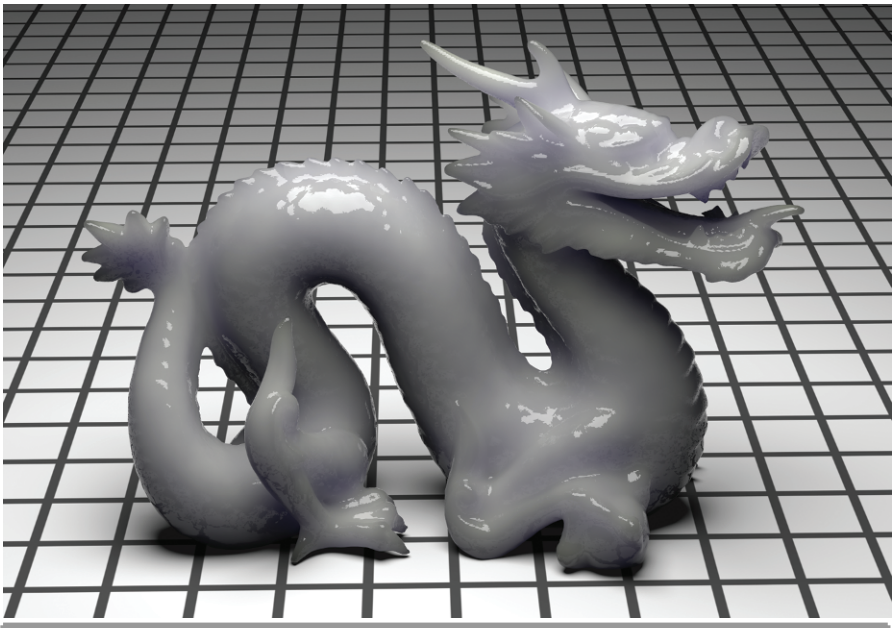


Figure 16.9: Subsurface Scattering from the Dragon Model. Although incident illumination is arriving from behind the model, the front of the model has light exiting from it, due to subsurface light transport.

3. At rendering time, these points are used in a hierarchical integration algorithm that evaluates the subsurface scattering equation at the point being shaded. To evaluate the integrand, it uses an approximation based on the dipole diffusion approximation that efficiently evaluates the BSSRDF for a given pair of points on the surface.

The `DipoleSubsurfaceIntegrator` class implements this approach. It is implemented in the files `integrators/dipolesubsurface.h` and `integrators/dipolesubsurface.cpp`. Its constructor takes parameters that set the maximum depth for specular reflections and refractions, a maximum error control (its use will be explained later in this section), a minimum distance that controls the density of points on the surfaces of objects for the first step above, and an optional filename. If the filename isn't an empty string, then a collection of precomputed points on (presumably translucent) surfaces is read from the given file. Otherwise, a set of points is found during a preprocessing step.

⟨*DipoleSubsurfaceIntegrator Public Methods*⟩ ≡

```
DipoleSubsurfaceIntegrator(int mdepth, float merror, float mindist,
                           const string &fn) {
    maxSpecularDepth = mdepth;
    maxError = merror;
    minSampleDist = mindist;
    filename = fn;
}
```

⟨*DipoleSubsurfaceIntegrator Private Data*⟩ ≡
```
int maxSpecularDepth;
float maxError, minSampleDist;
string filename;
```

16.5.1 POISSON DISTRIBUTION OF SAMPLE POINTS

For the hierarchical integration algorithm, we need to generate a collection of points on the surfaces of the translucent objects in the scene at which we'll compute incident irradiance. These points should be reasonably uniformly distributed so that they sample the irradiance at a regular rate; if there were clumps of multiple points close together, the hierarchical integration algorithm would compute poor results.

A Poisson disk distribution of points on the surface would be a good distribution for integration. (Recall the 2D Poisson disk sampling pattern for images in Section 7.5, though there the points weren't on object surfaces.) A number of approaches to directly compute such a distribution are available, though all require a more complex set of `Shape` interfaces than `pbrt` provides. (For example, if a parameterization of the surface is available or if the geometry is constrained to be a triangle mesh, *point repulsion* algorithms can be used, where a number of points are placed randomly on the surface and then their positions are iteratively adjusted until none of them are too close together.) Rather than requiring `Shapes` to provide methods to support algorithms like these, which would make adding new `Shapes` to the system more difficult, we will instead implement an approach that places no new requirements on `Shapes`.

In the implementation here, we first generate random candidate points on the surfaces of translucent objects. These points are then either accepted or rejected based on a Poisson sphere test: if no other accepted points are within a given 3D distance of the candidate point, the point is accepted. This process continues until a few thousand candidate tests have been rejected in a row, suggesting that no more will be found that pass the Poisson sphere test. The approximation of using a 3D Poisson sphere criterion to approximate a 2D Poisson disk on the surface of a complex object generally works well, though for some geometric configurations it can be significantly inaccurate; Figure 16.10 shows some examples. (These cases tend to be ones where the semi-infinite slab approximation breaks down as well.) Exercise 16.12 at the end of the chapter discusses these issues further and suggests a few approaches to improve this part of the implementation. Figure 16.11 shows the collection of points computed by this algorithm for the dragon model.

This task is handled by the `SurfacePointsRenderer`, defined in the files `renderers/surfacepoints.h` and `renderers/surfacepoints.cpp`. This functionality is provided as a `Renderer` so that point sets can be precomputed and stored in a file for later use with the `DipoleSubsurfaceIntegrator`.

⟨*SurfacePointsRenderer Public Methods*⟩ ≡
```
SurfacePointsRenderer(float md, const Point &pc, float t,
                      const string &fn)
    : minDist(md), time(t), pCamera(pc), filename(fn) { }
```

DipoleSubsurfaceIntegrator 887
Point 63
Renderer 24
Shape 108

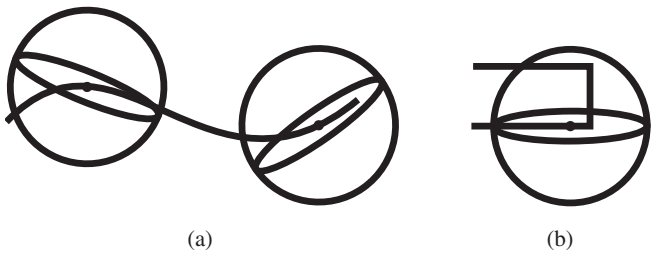


Figure 16.10: Cases Where the Poisson Sphere Criterion Works Well and Where It Fails. (a) For this relatively flat surface, distributing points with a Poisson sphere test gives a result similar to an actual Poisson disk test on the surface. (b) For this surface, with corners and a portion that folds over, the Poisson sphere criterion is too conservative, preventing points from being stored in some areas even though a Poisson disk test on the surface would include them.

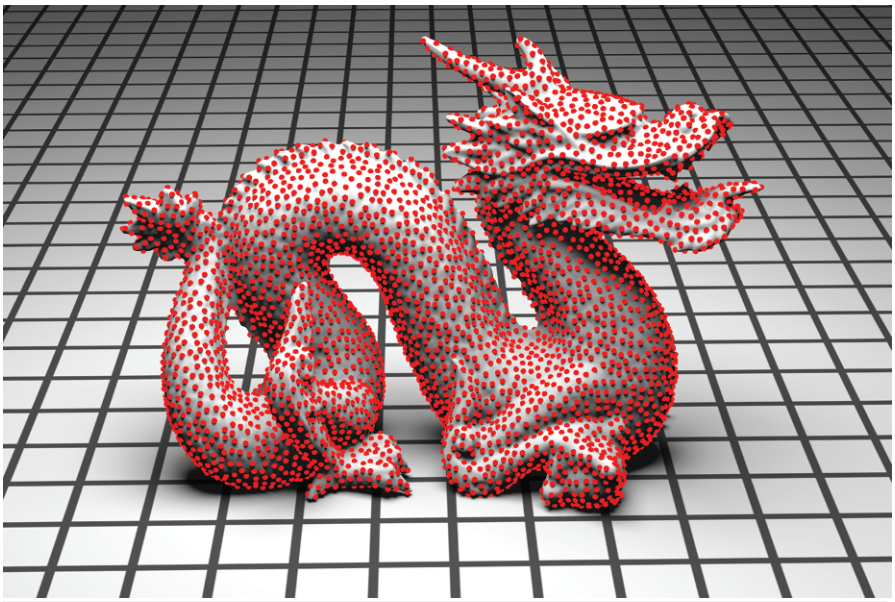


Figure 16.11: Sampled Points on the Dragon Model. This image shows a collection of points generated on the dragon model's surface using the point distribution algorithm implemented in this section.

⟨*SurfacePointsRenderer Private Data*⟩ ≡

```
float minDist, time;
Point pCamera;
string filename;
```

Point 63

To implement the Poisson sphere approach, we need to be able to generate an arbitrary number of random candidate points on translucent scene surfaces, and we need to

have an efficient way to reject the candidate points that don't fulfill the Poisson sphere requirement. To generate candidate points, we use a random sampling algorithm based on tracing rays through the scene and choosing a random scattered direction at each intersection point. This point sampling algorithm was introduced by Lehtinen et al. (2008) (though it wasn't applied to subsurface scattering). An octree is then used to store the accepted points, enabling efficient Poisson sphere tests for candidates. Because this funcionality is implemented in the context of a `Renderer` implementation, this work is done in the `Render()` method.

⟨*SurfacePointsRenderer Method Definitions*⟩ ≡
```
void SurfacePointsRenderer::Render(const Scene *scene) {
    ⟨Declare shared variables for Poisson point generation 890⟩
    ⟨Launch tasks to trace rays to find Poisson points⟩
    if (filename != "") {
        ⟨Write surface points to file⟩
    }
}
```

A number of common variables are needed by the tasks that will be computing these points. First, the tasks use a single shared `Octree` to store the accepted points and a few integer variables that record totals of the number of points that have been accepted, the number of points that have been rejected, and so forth. Once `maxFails` candidate points in a row have been rejected, the sampling algorithm terminates.

Due to the presence of these shared variables (which are periodically modified by the tasks), the tasks do require some synchronization to check and update their values; they use a single `RWMutex` for this task.

⟨*Declare shared variables for Poisson point generation*⟩ ≡ 890
```
BBox octBounds = scene->WorldBound();
octBounds.Expand(.001f * powf(octBounds.Volume(), 1.f/3.f));
Octree<SurfacePoint> pointOctree(octBounds);
⟨Create scene bounding sphere to catch rays that leave the scene 891⟩
int maxFails = 2000, repeatedFails = 0, maxRepeatedFails = 0;
int totalPathsTraced = 0, totalRaysTraced = 0, numPointsAdded = 0;
```

The accepted points are stored in an array of `SurfacePoints`. They are stored in a `SurfacePointsRenderer` member variable so that they persist after the `Render()` method exits.

⟨*SurfacePointsRenderer Private Data*⟩ +≡
```
vector<SurfacePoint> points;
```

⟨*SurfacePointsRenderer Declarations*⟩ ≡
```
struct SurfacePoint {
    SurfacePoint() { }
    SurfacePoint(const Point &pp, const Normal &nn, float a, float eps)
        : p(pp), n(nn), area(a), rayEpsilon(eps) { }
    ⟨SurfacePoint Data 891⟩
};
```

BBox 70
BBox::Expand() 72
BBox::Volume() 72
Normal 65
Octree 1023
Point 63
Renderer 24
RWMutex 1039
Scene 22
Scene::WorldBound() 24
SurfacePoint 890

⟨*SurfacePoint Data*⟩ ≡ 890
```
Point p;
Normal n;
float area, rayEpsilon;
```

The implementation uses a `GeometricPrimitive` that holds a sphere that surrounds the entire scene. For scenes that aren't closed, the point distribution algorithm needs an object like this that "catches" any rays that would otherwise leave the scene so that they can be redirected back into the scene.

⟨*Create scene bounding sphere to catch rays that leave the scene*⟩ ≡ 890, 962
```
Point sceneCenter;
float sceneRadius;
scene->WorldBound().BoundingSphere(&sceneCenter, &sceneRadius);
Transform ObjectToWorld(Translate(sceneCenter - Point(0,0,0)));
Transform WorldToObject(Inverse(ObjectToWorld));
Reference<Shape> sph = new Sphere(&ObjectToWorld, &WorldToObject,
    true, sceneRadius, -sceneRadius, sceneRadius, 360.f);
Reference<Material> nullMaterial = Reference<Material>(NULL);
GeometricPrimitive sphere(sph, nullMaterial, NULL);
```

`Tasks` are launched by the fragment ⟨*Launch tasks to trace rays to find Poisson points*⟩ (not included here) to generate the Poisson sphere points in parallel. The `SurfacePointTask::Run()` method does the computation. The computation in this method goes through three phases in turn:

1. Random ray paths are followed through the scene; at each intersection point, if the surface is translucent, then a candidate point is stored in local memory owned by the task. For all surfaces, a uniformly sampled direction is then used for the outgoing direction of a new ray starting at the intersection point. Note that this step requires just read-only access to the scene description.
2. After a few thousand ray paths have been followed and candidate points have been stored, the task acquires a reader lock on the reader–writer mutex that protects the octree. The task then checks all of the candidate points to see if there is an accepted point already in the octree that's too close to the candidate point, violating the Poisson sphere condition.
3. For the candidate points that pass the first test, the task then acquires a writer lock of the mutex, ensuring that no other thread will either read from or write to the octree during this time. The remaining points are checked again with the contents of the octree and the valid ones are added to it.

This process repeats until a large number of candidate points have been rejected in a row; at this point, the tasks exit under the assumption that no more valid points will be found.

This process was designed to minimize the amount of time that tasks held the writer lock, thus excluding all of other tasks from accessing the octree at all. Note that the second phase of the approach described above could be eliminated without affecting the correctness of the algorithm, though this would mean that tasks held the writer lock for

BBox::BoundingSphere() 74
GeometricPrimitive 188
Material 483
Normal 65
Point 63
Reference 1011
Scene::WorldBound() 24
Shape 108
Sphere 115
SurfacePointTask::Run() 892
Task 1041
Transform 76
Translate() 79

longer; if they can instead reject as many points as possible while using only read access, then they will spend less time holding the writer lock, and other tasks can potentially simultaneously test their candidate points using read access as well.

It is important to retest the points that passed the Poisson sphere test in the second step while actually holding the writer lock in the third step. Not only does the first test not compare the points with the other candidate points generated by the same task, but it's possible that another task may have acquired the writer lock and updated the tree between the time that the task does the first point tests with the read-only octree and the time that the task then adds the points in the read-write octree.

```
⟨SurfacePointsRenderer Method Definitions⟩ +≡
  void SurfacePointTask::Run() {
      ⟨Declare common variables for SurfacePointTask::Run() 892⟩
      while (true) {
          int pathsTraced, raysTraced = 0;
          for (pathsTraced = 0; pathsTraced < 20000; ++pathsTraced) {
              ⟨Follow ray path and attempt to deposit candidate sample points 893⟩
          }
          ⟨Make first pass through candidate points with reader lock 894⟩
          ⟨Make second pass through points with writer lock and update octree 895⟩
          ⟨Stop following paths if not finding new points 896⟩
          candidates.erase(candidates.begin(), candidates.end());
      }
  }
```

A few variables are initialized at the start of the task. The random number generator is seeded based on the number of the task currently running, ensuring that different tasks use different sequences of pseudo-random numbers. The `candidates` array stores the candidate points generated by the ray-tracing algorithm.

```
⟨Declare common variables for SurfacePointTask::Run()⟩ ≡ 892
  RNG rng(37 * taskNum);
  MemoryArena arena;
  vector<SurfacePoint> candidates;
```

The candidate point distribution algorithm needs a point in the scene from which to start the ray paths. Paths from that point should be able to reach the surfaces of all translucent objects in the scene. In particular, this means that the point should not be inside a wall or other closed object in the scene. Here, the paths start from the camera position, which is passed to the `SurfacePointTask` constructor. This works well for most scenes, though there are reasonable scenes for which it doesn't work well; consider a translucent object seen through a glass window—paths starting from the camera will never go through the window, since the path generation algorithm here samples new directions in the same hemisphere as the incident ray.

Another alternative would be to allow multiple starting points to be provided, or to randomly select starting points from inside the overall scene bounding box. However,

MemoryArena 1015
RNG 1003
SurfacePoint 890

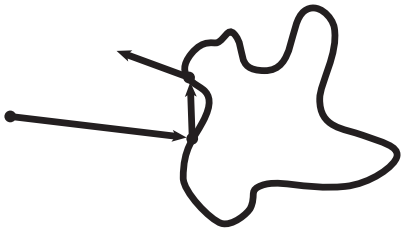


Figure 16.12: The Need for a Scene Bounding Sphere. If the scene is only comprised of a single triangle mesh, for example, and if all ray paths start from a single point (filled circle on left), then we need a sphere that bounds the scene for rays to intersect so that they can find paths to the backside of the object with respect to the path starting point.

selecting random points in the scene bounds would be inefficient in an extremely complex model like an office building where the camera was just looking at the contents of a single room; few of the paths would pass through the visible scene as seen by the camera.

The first few ray intersections along this path are ignored, as we'd like to generate a reasonably uniform distribution of random candidate points. (It's not critical that the distribution of candidate points be random, only that the ray paths reach all of the translucent objects in the scene.) For all rays that do hit a translucent object (i.e., one with a non-`NULL` BSSRDF), the point and some of its local geometric information are stored in the `candidates` array.

⟨*Follow ray path and attempt to deposit candidate sample points*⟩ ≡ 892

```
Vector dir = UniformSampleSphere(rng.RandomFloat(), rng.RandomFloat());
Ray ray(origin, dir, 0.f, INFINITY, time);
while (ray.depth < 30) {
    ⟨Find ray intersection with scene geometry or bounding sphere 894⟩
    ⟨Store candidate sample point at ray intersection if appropriate 894⟩
    ⟨Generate random ray from intersection point 894⟩
}
arena.FreeAll();
```

If the current ray doesn't hit any scene geometry, we test for intersection with the sphere that bounds the scene. Figure 16.12 shows why this sphere is needed; consider a scene composed of just a single translucent triangle mesh. With ray paths starting from a single point, multiple bounces between points on the mesh will not in general be able to reach some parts of the model's surface.

After a hit is found, the surface normal at the hit point is reoriented if needed so that it faces on the side of the surface that the ray arrived from.

`INFINITY` 1002
`MemoryArena::FreeAll()` 1017
`Ray` 66
`Ray::depth` 67
`RNG::RandomFloat()` 1003
`UniformSampleSphere()` 664
`Vector` 57

⟨*Find ray intersection with scene geometry or bounding sphere*⟩ ≡ 893
```
++raysTraced;
Intersection isect;
bool hitOnSphere = false;
if (!scene->Intersect(ray, &isect)) {
    if (!sphere.Intersect(ray, &isect))
        break;
    hitOnSphere = true;
}
DifferentialGeometry &hitGeometry = isect.dg;
hitGeometry.nn = Faceforward(hitGeometry.nn, -ray.d);
```

For points on translucent surfaces that have undergone a few bounces in the path, a `SurfacePoint` is added to the candidate array. The area on the surface that the point's sample represents is approximated with the area of the circle from the Poisson criterion. This approximation is accurate for reasonably flat surfaces and works well in practice.

⟨*Store candidate sample point at ray intersection if appropriate*⟩ ≡ 893
```
if (!hitOnSphere && ray.depth >= 3 &&
    isect.GetBSSRDF(RayDifferential(ray), arena) != NULL) {
    float area = M_PI * minSampleDist * minSampleDist;
    candidates.push_back(SurfacePoint(hitGeometry.p, hitGeometry.nn,
                                      area, isect.rayEpsilon));
}
```

At the intersection point, a uniform direction is sampled in the hemisphere around the surface normal at the point so that the new ray leaves the surface on the same side from which the previous one arrived. In general, we don't want the rays to get caught inside closed objects in the scene (like walls), though as noted above this choice means that we may miss sampling some parts of the scene that are visible through transparent objects.

⟨*Generate random ray from intersection point*⟩ ≡ 893
```
Vector dir = UniformSampleSphere(rng.RandomFloat(), rng.RandomFloat());
dir = Faceforward(dir, hitGeometry.nn);
ray = Ray(hitGeometry.p, dir, ray, isect.rayEpsilon);
```

After a number of paths have been followed and, presumably, candidate points generated, the second phase starts, with the read lock held. We'd like to reject any points that we can while only holding a read lock, thus reducing the time a write lock is held later.

⟨*Make first pass through candidate points with reader lock*⟩ ≡ 892
```
vector<bool> candidateRejected;
RWMutexLock lock(mutex, READ);
for (uint32_t i = 0; i < candidates.size(); ++i) {
    PoissonCheck check(minSampleDist, candidates[i].p);
    octree.Lookup(candidates[i].p, check);
    candidateRejected.push_back(check.failed);
}
```

The `PoissonCheck` helper class is called by the octree for all of the points that it stores that could be within the minimum distance of the point passed to its `Lookup()` method. If it

DifferentialGeometry 102
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
Faceforward() 66
GeometricPrimitive::
Intersect() 188
Intersection 186
Intersection::
GetBSSRDF() 484
Intersection::rayEpsilon 186
M_PI 1002
Octree::Lookup() 1027
PoissonCheck 895
Ray 66
RayDifferential 69
RNG::RandomFloat() 1003
RWMutexLock 1039
Scene::Intersect() 23
SurfacePoint 890
UniformSampleSphere() 664
Vector 57

finds any point closer than the maximum distance, it updates `failed` and returns `false`, indicating to the octree that it should stop any further traversal.

⟨*SurfacePointsRenderer Local Declarations*⟩ ≡
```
struct PoissonCheck {
    PoissonCheck(float md, const Point &pt)
        { maxDist2 = md * md; failed = false; p = pt; }
    float maxDist2;
    bool failed;
    Point p;
    bool operator()(const SurfacePoint &sp) {
        if (DistanceSquared(sp.p, p) < maxDist2) {
            failed = true; return false;
        }
        return true;
    }
};
```

The task now takes a write lock; it can now update the shared variables and octree knowing that no other task will be reading or writing any of them concurrently. The `repeatedFails` variable records how many candidate points in a row have been rejected. Another task may have hit the limit, in which case all of the other tasks exit the next time they acquire the write lock and discover that candidate generation should stop.

⟨*Make second pass through points with writer lock and update octree*⟩ ≡ 892
```
lock.UpgradeToWrite();
if (repeatedFails >= maxFails)
    return;
totalPathsTraced += pathsTraced;
totalRaysTraced += raysTraced;
for (uint32_t i = 0; i < candidates.size(); ++i) {
    if (candidateRejected[i]) {
        ⟨Update for rejected candidate point 895⟩
    }
    else {
        ⟨Recheck candidate point and possibly add to octree 896⟩
    }
}
```

For each rejected point, `repeatedFails` is incremented; when it hits the maximum, the task exits.

⟨*Update for rejected candidate point*⟩ ≡ 895, 896
```
++repeatedFails;
if (repeatedFails >= maxFails)
    return;
```

For points that passed the first check, we need to verify that they still pass the Poisson sphere test. As mentioned previously, the effect of updates to the octree from other tasks or from other points in this task's set of candidates means that the candidate point may

DistanceSquared() 65
IrradiancePoint::p 898
Point 63
PoissonCheck 895
RWMutexLock::
  UpgradeToWrite() 1039
SurfacePoint 890

not in fact still be valid. If it again passes, the octree is updated and the point is added to surfacePoints, which is a member variable of the SurfacePointsRenderer class passed into the tasks by reference.

⟨*Recheck candidate point and possibly add to octree*⟩ ≡ 895
```
SurfacePoint &sp = candidates[i];
PoissonCheck check(minSampleDist, sp.p);
octree.Lookup(sp.p, check);
if (check.failed) {
    ⟨Update for rejected candidate point 895⟩
}
else {
    ++numPointsAdded;
    repeatedFails = 0;
    Vector delta(minSampleDist, minSampleDist, minSampleDist);
    octree.Add(sp, BBox(sp.p-delta, sp.p+delta));
    surfacePoints.push_back(sp);
}
```

Finally, it's possible that many paths will be traced but no translucent objects are found. In this case, a warning is printed and the tasks exit.

⟨*Stop following paths if not finding new points*⟩ ≡ 892
```
if (totalPathsTraced > 50000 && numPointsAdded == 0) {
    Warning("There don't seem to be any objects with BSSRDFs "
            "in this scene.  Giving up.");
    return;
}
```

After the tasks have exited, the points, their surface normals, their corresponding areas, and ray epsilon values are written to a file if a filename was provided. This is handled by the ⟨*Write surface points to file*⟩ fragment, which is straightforward and thus not included here.

For the convenience of the DipoleSubsurfaceIntegrator, a short utility function can be called to compute a surface point set for a scene without the caller needing to explicitly create a SurfacePointsRenderer.

⟨*SurfacePointsRenderer Method Definitions*⟩ +≡
```
void FindPoissonPointDistribution(const Point &pCamera, float time,
        float minDist, const Scene *scene, vector<SurfacePoint> *points) {
    SurfacePointsRenderer sp(minDist, pCamera, time, "");
    sp.Render(scene);
    points->swap(sp.points);
}
```

BBox 70
DipoleSubsurfaceIntegrator 887
Octree::Add() 1024
Octree::Lookup() 1027
Point 63
PoissonCheck 895
Scene 22
SurfacePoint 890
SurfacePointsRenderer 888
Vector 57

16.5.2 BUILDING THE SAMPLE POINT OCTREE

We can now turn to the implementation of the `Preprocess()` method of the `DipoleSubsurfaceIntegrator`. It starts by getting a collection of `SurfacePoints` on the surfaces of translucent objects, either loading them from a file generated by the `SurfacePointsRenderer` or generating them here. Incident irradiance is then computed at each point, and then an octree that encodes a hierarchical clustering of the points is created.

⟨*DipoleSubsurfaceIntegrator Method Definitions*⟩ ≡
```
void DipoleSubsurfaceIntegrator::Preprocess(const Scene *scene,
        const Camera *camera, const Renderer *renderer) {
    if (scene->lights.size() == 0) return;
    vector<SurfacePoint> pts;
    ⟨Get SurfacePoints for translucent objects in scene 897⟩
    ⟨Compute irradiance values at sample points 897⟩
    ⟨Create octree of clustered irradiance samples 898⟩
}
```

⟨*Get* `SurfacePoints` *for translucent objects in scene*⟩ ≡ 897
```
if (filename != "") {
    ⟨Initialize SurfacePoints from file⟩
}
if (pts.size() == 0) {
    Point pCamera = camera->CameraToWorld(camera->shutterOpen,
                                          Point(0, 0, 0));
    FindPoissonPointDistribution(pCamera, camera->shutterOpen,
                                 minSampleDist, scene, &pts);
}
```

For each `SurfacePoint`, the incident irradiance from the light sources in the scene is computed. An `IrradiancePoint` is then created for each point.

⟨*Compute irradiance values at sample points*⟩ ≡ 897
```
RNG rng;
MemoryArena arena;
for (uint32_t i = 0; i < pts.size(); ++i) {
    SurfacePoint &sp = pts[i];
    Spectrum E(0.f);
    for (uint32_t j = 0; j < scene->lights.size(); ++j) {
        ⟨Add irradiance from light at point⟩
    }
    irradiancePoints.push_back(IrradiancePoint(sp, E));
    arena.FreeAll();
}
```

⟨*DipoleSubsurfaceIntegrator Private Data*⟩ +≡
```
vector<IrradiancePoint> irradiancePoints;
```

Camera 302
Camera::CameraToWorld 302
Camera::shutterOpen 302
FindPoissonPoint Distribution() 896
IrradiancePoint 898
MemoryArena 1015
MemoryArena::FreeAll() 1017
Point 63
Renderer 24
RNG 1003
Scene 22
Scene::lights 23
Spectrum 263
SurfacePoint 890
SurfacePointsRenderer 888

⟨*DipoleSubsurfaceIntegrator Helper Declarations*⟩ ≡
```
struct IrradiancePoint {
    IrradiancePoint(const SurfacePoint &sp, const Spectrum &ee)
        : p(sp.p), n(sp.n), E(ee), area(sp.area),
          rayEpsilon(sp.rayEpsilon) { }
    Point p;
    Normal n;
    Spectrum E;
    float area, rayEpsilon;
};
```

The implementation of the fragment ⟨*Add irradiance from light at point*⟩ is straightforward and not included here; it computes the contribution of direct lighting to irradiance, ignoring indirect illumination, taking a number of samples for each light to estimate the integral from Equation (5.3). This computation is currently done in a single thread of execution; it is usually fast enough that breaking it into tasks is of limited value. For many applications, it is desirable to instead compute irradiance including indirect illumination. If the indirect illumination computation is more computationally intensive, parallelization of this step may be appropriate.

After the irradiance values for the `IrradiancePoints` are computed, the octree is created. The `Preprocess()` method allocates the root node of the tree, computes the bounding box of all of the points, and inserts all of the points into the octree in turn. The root node starts out as an empty leaf node but progressively refines itself to have children nodes (which in turn refine themselves) as more points are added to it.

⟨*Create octree of clustered irradiance samples*⟩ ≡ 897
```
octree = octreeArena.Alloc<SubsurfaceOctreeNode>();
for (uint32_t i = 0; i < irradiancePoints.size(); ++i)
    octreeBounds = Union(octreeBounds, irradiancePoints[i].p);
for (uint32_t i = 0; i < irradiancePoints.size(); ++i)
    octree->Insert(octreeBounds, &irradiancePoints[i], octreeArena);
octree->InitHierarchy();
```

⟨*DipoleSubsurfaceIntegrator Private Data*⟩ +≡
```
BBox octreeBounds;
SubsurfaceOctreeNode *octree;
MemoryArena octreeArena;
```

The `SubsurfaceOctreeNode` class represents a node of the octree that stores the hierarchical clustered irradiance points.

⟨*DipoleSubsurfaceIntegrator Local Declarations*⟩ ≡
```
struct SubsurfaceOctreeNode {
    ⟨SubsurfaceOctreeNode Methods 899⟩
    ⟨SubsurfaceOctreeNode Public Data 899⟩
};
```

BBox 70
DipoleSubsurfaceIntegrator::
irradiancePoints 897
DipoleSubsurfaceIntegrator::
octree 898
DipoleSubsurfaceIntegrator::
octreeArena 898
IrradiancePoint 898
MemoryArena 1015
MemoryArena::Alloc() 1016
Normal 65
Point 63
Spectrum 263
SubsurfaceOctreeNode 898
SubsurfaceOctreeNode::
InitHierarchy() 901
SubsurfaceOctreeNode::
Insert() 899
SurfacePoint 890
Union() 72

The leaf nodes of the octree hold pointers to up to eight `IrradiancePoints`. Interior nodes hold pointers to the eight potential children. This octree follows the conventions of the `Octree` template class in Section A.7 in terms of the ordering of the children. Each node of the octree also holds aggregate information about the `IrradiancePoints` in the tree beneath it, including the average position and irradiance of the points underneath it as well as the sum of their areas.

⟨*SubsurfaceOctreeNode Public Data*⟩ ≡ 898
```
Point p;
bool isLeaf;
Spectrum E;
float sumArea;
union {
    SubsurfaceOctreeNode *children[8];
    IrradiancePoint *ips[8];
};
```

⟨*SubsurfaceOctreeNode Methods*⟩ ≡ 898
```
SubsurfaceOctreeNode() {
    isLeaf = true;
    sumArea = 0.f;
    for (int i = 0; i < 8; ++i)
        ips[i] = NULL;
}
```

The `Insert()` method first computes the midpoint of the current node, which will be used by all code paths later in the method, and then determines what to do with the point being added. (It is a precondition that the point is inside the current node.) If the node is a leaf, then the pointer to the point is added to the points stored in the leaf; otherwise the `Insert()` method of the child node that the point overlaps will be called, progressing recursively until a leaf node is reached.

⟨*SubsurfaceOctreeNode Methods*⟩ +≡ 898
```
void Insert(const BBox &nodeBound, IrradiancePoint *ip,
            MemoryArena &arena) {
    Point pMid = .5f * nodeBound.pMin + .5f * nodeBound.pMax;
    if (isLeaf) {
        ⟨Add IrradiancePoint to leaf octree node 900⟩
    }
    ⟨Add IrradiancePoint ip to interior octree node 900⟩
}
```

Leaf nodes can only store eight `IrradiancePoints`; the pointers for them start out all `NULL` and are initialized in turn. If all of the pointers are in use when a new point is added, then the node converts itself to an interior node and distributes its `IrradiancePoints` to newly allocated children nodes.

BBox 70
BBox::pMax 71
BBox::pMin 71
IrradiancePoint 898
MemoryArena 1015
Octree 1023
Point 63
Spectrum 263
SubsurfaceOctreeNode 898
SubsurfaceOctreeNode::ips 899
SubsurfaceOctreeNode::isLeaf 899
SubsurfaceOctreeNode::sumArea 899

⟨*Add* `IrradiancePoint` *to leaf octree node*⟩ ≡ 899
```
for (int i = 0; i < 8; ++i) {
    if (!ips[i]) {
        ips[i] = ip;
        return;
    }
}
⟨Convert leaf node to interior node, redistribute points 900⟩
/* fall through to interior case to insert the new point... */
```

To convert a leaf to an interior node, the `IrradiancePoint` pointers first must be copied to an array separate from the one in the node. Recall that the pointers to the `IrradiancePoints` share a `union` with the pointers to the children; if the pointers weren't copied first, then they'd be clobbered as we allocated new children nodes and set their pointers in the code below. Even worse, the implementation would interpret `IrradiancePoint` pointers as `SubsurfaceOctreeNode` pointers, leading to major errors or crashes.

After the points are copied, the `children` pointers are set to `NULL` and the `IrradiancePoint` pointers are inserted into newly allocated children of this node.

⟨*Convert leaf node to interior node, redistribute points*⟩ ≡ 900
```
isLeaf = false;
IrradiancePoint *localIps[8];
for (int i = 0; i < 8; ++i) {
    localIps[i] = ips[i];
    children[i] = NULL;
}
for (int i = 0; i < 8; ++i)  {
    IrradiancePoint *ip = localIps[i];
    ⟨Add IrradiancePoint ip to interior octree node 900⟩
}
```

The following fragment is used twice; once in the `for` loop in the fragment immediately above here, and once in the definition of `SubsurfaceOctreeNode::Insert()`.

⟨*Add* `IrradiancePoint ip` *to interior octree node*⟩ ≡ 899, 900
```
int child = (ip->p.x > pMid.x ? 4 : 0) +
    (ip->p.y > pMid.y ? 2 : 0) + (ip->p.z > pMid.z ? 1 : 0);
if (!children[child])
    children[child] = arena.Alloc<SubsurfaceOctreeNode>();
BBox childBound = octreeChildBound(child, nodeBound, pMid);
children[child]->Insert(childBound, ip, arena);
```

Once all of the points have been added to the octree, `InitHierarchy()` is called. This method computes the aggregate values at each node: the average of the positions and irradiance values and the sum of the areas of the children.

BBox 70
IrradiancePoint 898
IrradiancePoint::p 898
MemoryArena::Alloc() 1016
octreeChildBound() 1026
SubsurfaceOctreeNode 898
SubsurfaceOctreeNode::children 899
SubsurfaceOctreeNode::Insert() 899
SubsurfaceOctreeNode::ips 899
SubsurfaceOctreeNode::isLeaf 899

⟨*SubsurfaceOctreeNode Methods*⟩ +≡ 898
```
void InitHierarchy() {
    if (isLeaf) {
        ⟨Init SubsurfaceOctreeNode leaf from IrradiancePoints 901⟩
    }
    else {
        ⟨Init interior SubsurfaceOctreeNode⟩
    }
}
```

At leaf nodes, the average irradiance and the total area that the points represent are both easily computed from the individual `IrradiancePoints`. The representative position for the samples is computed as a luminance-weighted average of the positions of the points; the idea is that points with relatively high irradiance values should have more weight in computing the average position used by the forthcoming hierarchical integration algorithm.

⟨*Init* `SubsurfaceOctreeNode` *leaf from* `IrradiancePoints`⟩ ≡ 901
```
float sumWt = 0.f;
uint32_t i;
for (i = 0; i < 8; ++i) {
    if (!ips[i]) break;
    float wt = ips[i]->E.y();
    E += ips[i]->E;
    p += wt * ips[i]->p;
    sumWt += wt;
    sumArea += ips[i]->area;
}
if (sumWt > 0.f) p /= sumWt;
E /= i;
```

At interior nodes, the ⟨*Init interior* `SubsurfaceOctreeNode`⟩ fragment first computes aggregate values for the children nodes via recursive calls to `InitHierarchy()`. Then, it computes aggregate values directly from the children's values. The computation otherwise follows the form of the computation in the fragment ⟨*Init* `SubsurfaceOctreeNode` *leaf from* `IrradiancePoints`⟩ and so won't be included here.

DipoleSubsurfaceIntegrator 887
IrradiancePoint 898
IrradiancePoint::area 898
IrradiancePoint::E 898
IrradiancePoint::p 898
Spectrum::y() 273
SubsurfaceOctreeNode::E 899
SubsurfaceOctreeNode::ips 899
SubsurfaceOctreeNode::isLeaf 899
SubsurfaceOctreeNode::p 899
SubsurfaceOctreeNode::sumArea 899

16.5.3 THE DIPOLE DIFFUSION APPROXIMATION

We will give an overview of the diffusion approximation of the equation of transfer as well as the dipole method for solving it; together these lead to the efficient approach used for subsurface scattering in the `DipoleSubsurfaceIntegrator`. The full derivations of each of these are lengthy and complex; as such, we won't include them in the text here. See Chapter 5 of Donner's Ph.D. dissertation (2006) for a full derivation of both. The "Further Reading" section also has pointers to additional information on these topics.

A number of important ideas are used in the process of transforming the fully general equation of transfer to the diffusion equation, which can be approximately solved for subsurface scattering. The first is the *principle of similarity*, which says that for an

anisotropically scattering medium with a high albedo, the medium can instead be modeled as having an isotropic phase function with appropriately modified scattering and extinction coefficients. Light transport solutions computed based on the modified coefficients correspond well to those with the original coefficients and phase function, while allowing simplifications due to the assumption of isotropic scattering.

The principle of similarity is based on the observation that after many scattering events the distribution of light in media with high albedos becomes more and more uniformly directionally distributed regardless of the original illumination distribution or the anisotropy of the phase function. One way to see how this happens is to consider an expression derived by Yanovitskij (1997); it describes isotropization due to multiple scattering events from the Henyey–Greenstein phase function. He showed that the distribution of light that has been scattered n times is given by

$$p_n(\omega \to \omega') = \frac{1 - g^{2n}}{4\pi(1 + g^{2n} - 2g|g^{n-1}|\,(-\omega \cdot \omega')^{3/2})}.$$

As n grows large, this converges to the isotropic phase function, $1/4\pi$. Figure 16.13 plots this function for a few values of n. When coupled with the observation from the start of this section about how much of the light energy remains after tens or even hundreds of scattering events in high-albedo materials, one can see intuitively why it is reasonable to work with an isotropic phase function approximation for high-albedo media.

The *reduced scattering coefficient* is defined as $\sigma_s' = (1 - g)\sigma_s$, where g is the anisotropy parameter, and the *reduced extinction coefficient* is $\sigma_t' = \sigma_a + \sigma_s'$. These new coefficients account for the effect of using the isotropic phase function approximation. To understand the idea they embody, consider a strongly forward-scattering phase function, where $g \to 1$. With the actual phase function, at each scattering event the light will mostly continue in the same direction to its next scattering event. In this case, the value of the reduced scattering coefficient σ_s' is much smaller than σ_s, which means that light travels a larger distance in the medium before scattering; the medium is effectively approximated as being thinner, allowing light to travel further, having the same effect as a highly forward-scattering phase function.

Conversely, consider the case of $g \to -1$. In this case, at a scattering event the light will tend to scatter back in the direction it came from. But then the next time it scatters after that, it will generally reverse course again; it bounces back and forth without making very much forward progress. In this case, the reduced scattering coefficient is larger than the original scattering coefficient, indicating greater probability of scattering interaction. In other words, the medium is treated as being thicker than it actually is, which approximates the effect of light having relatively more trouble making forward progress. Figure 16.14 illustrates these ideas, showing representative paths of scattering interactions in highly forward-scattering and highly backward-scattering media.

The diffusion equation can be derived from the equation of transfer; it provides a solution to the equation of transfer for the case of homogeneous, optically thick, highly scattering materials (i.e., those with relatively large albedos). For the application to subsurface scattering in `pbrt`, it can be derived by writing the equation of transfer using the reduced scattering and extinction coefficients and an isotropic phase function. Then, the terms that account for single scattering and light transmitted directly through the me-

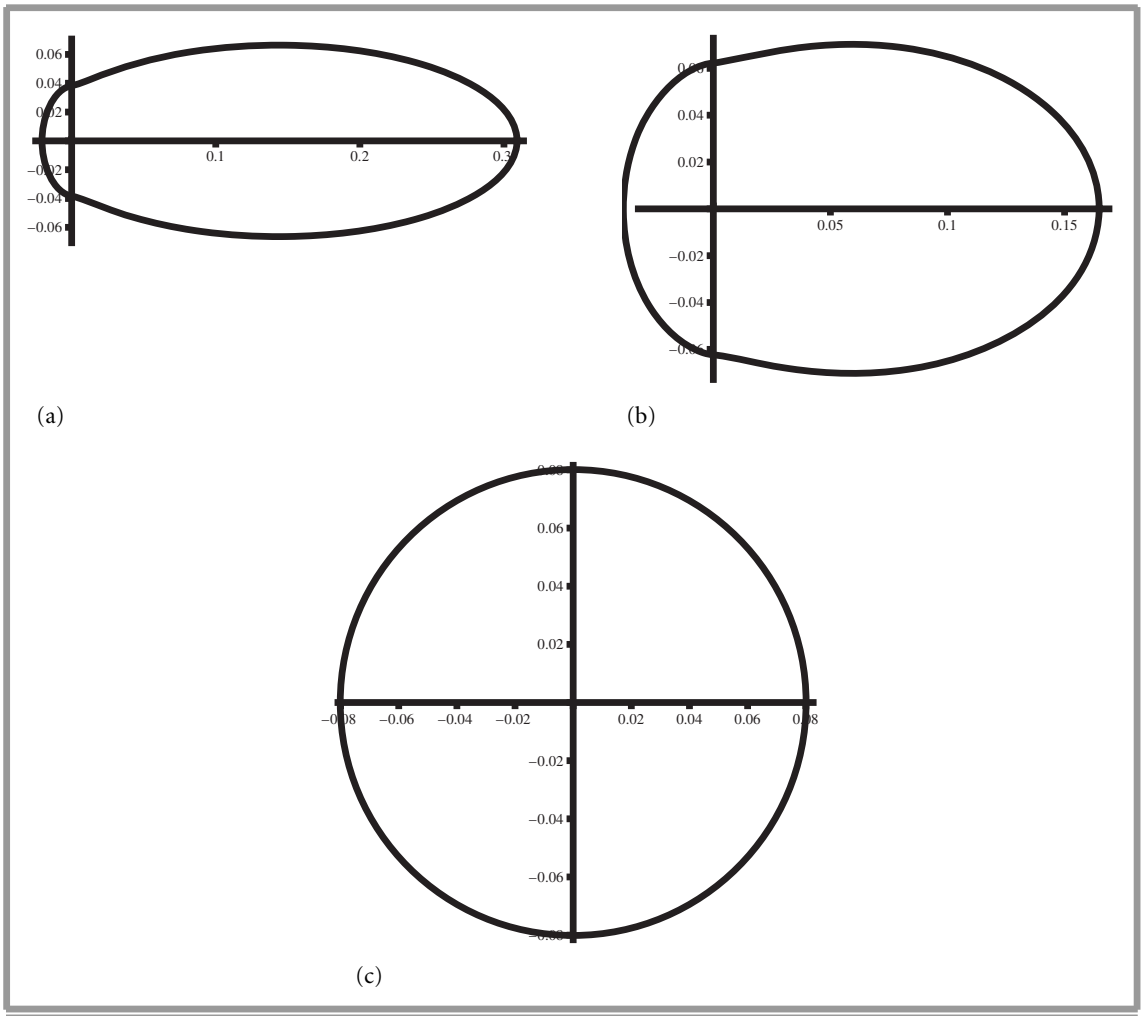


Figure 16.13: Light Distribution After Many Scattering Events. (a) Directional distribution of a single incident ray of light after 10 scattering events in a highly anisotropic medium with $g = 0.9$, (b) 100 scattering events, (c) 1000 scattering events. The distribution becomes increasingly isotropic, even though it was initially very anisotropic.

dium are removed, so that only multiple scattering is considered. (Those two terms are easily handled separately.)

The resulting spatially varying radiance function $L(\mathrm{p}, \omega)$ is expressed with four terms in the spherical harmonic basis (Section 17.2), essentially as a constant term plus linear functions of x, y, and z. The result is the diffusion equation,

$$\nabla \cdot \left(\frac{\nabla \phi(\mathrm{p})}{\sigma_{\mathrm{t}}'} \right) - 3\sigma_{\mathrm{a}} \phi(\mathrm{p}) = 0$$

Figure 16.14: Representative Light Paths for Highly Anisotropic Scattering Media. (a) Forward-scattering medium, with $g = 0.95$. Light generally scatters in the same direction it was originally traveling. (b) Backward-scattering medium, with $g = -0.95$. Light frequently bounces back and forth, making relatively little forward progress with respect to its original direction.

where ϕ represents *fluence*,

$$\phi(\mathrm{p}) = \int_{S^2} L_\mathrm{i}(\mathrm{p}, \omega)\, \mathrm{d}\omega.$$

The diffusion equation can be solved in certain conditions. For example, the fluence at a point $\mathrm{p_i}$ from a point light source source at point p with power Φ in an infinite medium has an analytic solution,

$$\phi(\mathrm{p}, \mathrm{p_i}) = \frac{\Phi}{4\pi D} \frac{\mathrm{e}^{-\sigma_\mathrm{tr}\|\mathrm{p}-\mathrm{p_i}\|}}{\|\mathrm{p} - \mathrm{p_i}\|} \qquad [16.3]$$

where $\sigma_\mathrm{tr} = \sqrt{3\sigma_\mathrm{t}'\sigma_\mathrm{a}}$, and D is the diffusion coefficient $D = 1/(3\sigma_\mathrm{t}')$.

For subsurface scattering, it's useful to consider the case of a semi-infinite slab: the scattering medium is all space below the $z = 0$ plane. To solve this diffusion equation in this case, a suitable boundary condition must be found. In particular, we'd like a boundary condition that lets us compute fluence on the boundary of the medium; it's not important to be able to compute the scattering at points inside of it, since we're only interested here in computing the light leaving it.

In the semi-infinite slab setting, there's no incident radiance at the top of the slab due to radiance from scattering within the slab. Thus, the incident radiance at the boundary in the hemisphere around the $(0, 0, -1)$ direction is equal to the incident radiance from scattering in the medium arriving at the boundary reflected by the Fresnel effect back

down into the medium. This boundary condition can be approximated as $\phi(2AD) = 0$, where

$$A = \frac{1 + F_{\mathrm{dr}}(\eta)}{1 - F_{\mathrm{dr}}(\eta)}$$

and F_{dr} is the diffuse Fresnel reflectance, given by

$$F_{\mathrm{dr}}(\eta) = \int_{\mathcal{H}^2(\mathbf{n})} F_{\mathrm{r}}(\eta, \omega_{\mathrm{i}})\, |\omega_{\mathrm{i}} \cdot \mathbf{n}|\, \mathrm{d}\omega_{\mathrm{i}}. \tag{16.4}$$

This boundary condition says that fluence goes to zero at a distance $2AD$ above the slab's boundary; it represents the extrapoloation of the fall-off of fluence inside the medium as the boundary is approached and thus leads to a reasonable solution for points on the actual boundary of the slab. Note that in general a point outside the medium at that distance would expect to have nonzero fluence due to radiance leaving the medium; however, our goal here is to compute a good solution at the boundary of the medium.

A short utility function computes the diffuse Fresnel reflectance using a closed-form rational approximation to Equation (16.4).

⟨*Reflection Declarations*⟩ +≡

```
inline float Fdr(float eta) {
    if (eta >= 1)
        return -1.4399f / (eta*eta) + 0.7099f / eta + 0.6681f +
            0.0636f * eta;
    else
        return -0.4399f + .7099f / eta - .3319f / (eta * eta) +
            .0636f / (eta*eta*eta);
}
```

One way to generate a fluence distribution that matches the boundary condition $\phi(2AD) = 0$ is to use the analytic solution for a point source, Equation (16.3), to compute the fluence for a pair of light sources, one with positive flux and one with negative flux. If these two light sources are placed appropriately, they will cancel out along the plane $z = 2AD$, fulfilling the boundary condition.

If light enters the medium at the point $\mathrm{p_i} = (x_{\mathrm{i}}, y_{\mathrm{i}}, 0)$ on the boundary, the *dipole approximation* uses two such sources to fulfill the boundary condition, with the positive point inside the medium at the point $(x_{\mathrm{i}}, y_{\mathrm{i}}, z^+)$ and the negative point above the medium at the point $(x_{\mathrm{i}}, y_{\mathrm{i}}, z^-)$ with $z^- = 2AD + z^+$. Thus, the fluence leaving the medium at a particular point $\mathrm{p_o} = (x_{\mathrm{o}}, y_{\mathrm{o}}, 0)$ is given by subtracting the contributions of the negative source from the contribution of the positive source:

$$\phi(\mathrm{p_i}, \mathrm{p_o}) = \frac{\Phi}{4\pi D}\left(\frac{\mathrm{e}^{-\sigma_{\mathrm{tr}} d^+}}{d^+} - \frac{\mathrm{e}^{-\sigma_{\mathrm{tr}} d^-}}{d^-}\right), \tag{16.5}$$

with the distance from the point to the sources given by

$$d^+ = \|(x_{\mathrm{i}}, y_{\mathrm{i}}, z^+) - \mathrm{p_o}\|, \text{ and } d^- = \|(x_{\mathrm{i}}, y_{\mathrm{i}}, z^-) - \mathrm{p_o}\|.$$

See Figure 16.15.

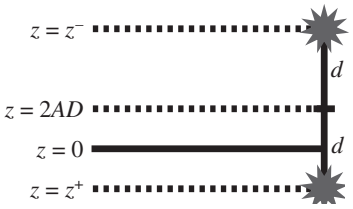


Figure 16.15: Basic Setting for the Dipole Approximation to the Solution to the Diffusion Equation. A light source with positive flux is placed at a position $z = z^+$ inside the medium, beneath the point where incident illumination arrives, and a second light with an equal amount of negative flux is placed at $z = z^-$ above the medium. These sources are placed so that the flux for the two of them cancels out at a height $z = 2AD$ above the boundary $z = 0$, fulfilling the desired boundary condition. The fluence that results from subtracting the closed-form expression of the fluence distributions for each of them (Equation (16.3)) at the medium's boundary $z = 0$ is a reasonable approximation of the fluence at the boundary due to subsurface scattering. If we use this approximation to compute the fluence at a point $\mathrm{p_o}$ on the boundary, we can find how much light is exiting the medium due to light entering at $\mathrm{p_i}$.

To compute the diffuse subsurface reflectance, we'd like to compute the radiant exitance, which is found from the fluence by taking the gradient of the fluence function and computing the dot product with the normal $\mathbf{n}$,

$$M_\mathrm{o}(\mathrm{p_o}, \mathbf{n}) = -D(\mathbf{n} \cdot \nabla\phi)(\mathrm{p_o}).$$

Equivalently, and more conveniently, we can take the derivative with respect to z,

$$M_\mathrm{o}(\mathrm{p_o}, \mathbf{n}) = -D\frac{\mathrm{d}}{\mathrm{d}z}\phi(\mathrm{p_o}).$$

The differential diffuse subsurface reflectance R_d due to illumination at a point $\mathrm{p_i}$ is given in turn by taking the differential $\mathrm{d}M_\mathrm{o}(\mathrm{p_o}, \mathbf{n})/\mathrm{d}\Phi(\mathrm{p_i})$.

Putting this all together and working through the derivative of Equation (16.5), we finally have the dipole approximation to diffuse subsurface reflectance, which we'll use for rendering in the implementation below.

$$R_\mathrm{d}(\mathrm{p_i}, \mathrm{p_o}) = \frac{1}{4\pi}\left(\frac{z^+(d^+\sigma_\mathrm{tr} + 1)\mathrm{e}^{-\sigma_\mathrm{tr}d^+}}{(d^+)^3} - \frac{z^-(d^-\sigma_\mathrm{tr} + 1)\mathrm{e}^{-\sigma_\mathrm{tr}d^-}}{(d^-)^3}\right). \quad [16.6]$$

Figure 16.16 is a graph of this function. Diffuse subsurface reflectance R_d is the analog of the reflectance of BSDFs ρ_hd and ρ_hh introduced in Section 8.1.1, in that it represents directionally averaged scattering.

`DiffusionReflectance` is a small utility class that computes the value of R_d using Equation (16.6). Rather than encapsulating this computation in a utility function, using a class makes it easier to precompute a number of common terms that only depend on the scattering properties of the medium. Their values can then be reused many times as R_d is evaluated for many different points.

`DiffusionReflectance` 907

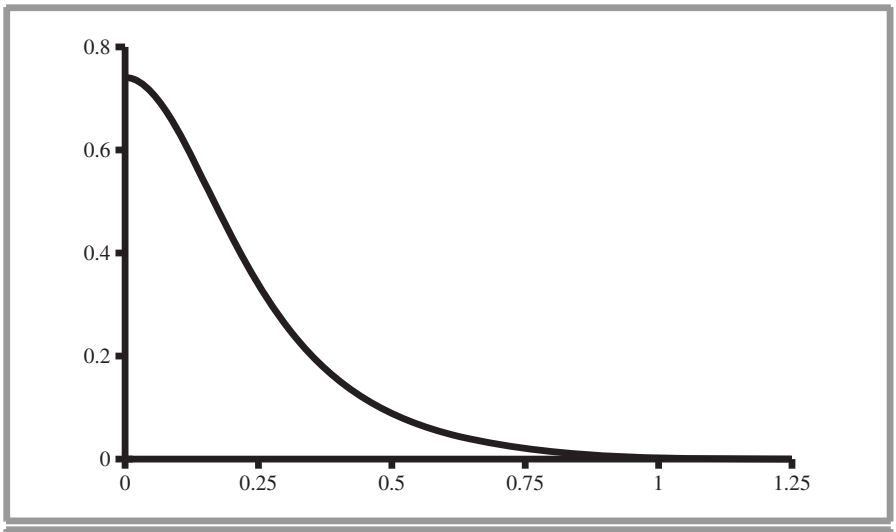


Figure 16.16: Graph of the R_d Function from Equation (16.6), Using Measured Scattering Properties of Milk. The value of R_d falls off rapidly as a function of distance d between the points p_i and p_o.

⟨*DipoleSubsurfaceIntegrator Local Declarations*⟩ +≡
```
struct DiffusionReflectance {
    ⟨DiffusionReflectance Public Methods 907⟩
    ⟨DiffusionReflectance Data 907⟩
};
```

The constructor precomputes the terms that are constant for all evaluations of R_d. Following Farrell et al. (1992), the positive source is placed at depth $z^+ = 1/\sigma_t'$ in the medium; this corresponds to the *mean free path*, the distance at which the first scattering event due to incident illumination at the top of the medium is expected to occur.

⟨*DiffusionReflectance Public Methods*⟩ ≡ 907
```
DiffusionReflectance(const Spectrum &sigma_a, const Spectrum &sigmap_s,
                     float eta) {
    A = (1.f + Fdr(eta)) / (1.f - Fdr(eta));
    sigmap_t = sigma_a + sigmap_s;
    sigma_tr = Sqrt(3.f * sigma_a * sigmap_t);
    alphap = sigmap_s / sigmap_t;
    zpos = Spectrum(1.f) / sigmap_t;
    zneg = zpos * (1.f + (4.f/3.f) * A);
}
```

DiffusionReflectance 907
Fdr() 905
Spectrum 263
Spectrum::Sqrt() 265

⟨*DiffusionReflectance Data*⟩ ≡ 907
```
Spectrum zpos, zneg, sigmap_t, sigma_tr, alphap;
float A;
```

The method to compute the reflectance takes the squared distance in the $z = 0$ plane between the point of incident illumination and the outgoing point of interest.

⟨*DiffusionReflectance Public Methods*⟩ +≡ 907
```
Spectrum operator()(float d2) const {
    Spectrum dpos = Sqrt(Spectrum(d2) + zpos * zpos);
    Spectrum dneg = Sqrt(Spectrum(d2) + zneg * zneg);
    Spectrum Rd = (1.f / (4.f * M_PI)) *
        ((zpos * (dpos * sigma_tr + Spectrum(1.f)) *
          Exp(-sigma_tr * dpos)) / (dpos * dpos * dpos) -
         (zneg * (dneg * sigma_tr + Spectrum(1.f)) *
          Exp(-sigma_tr * dneg)) / (dneg * dneg * dneg));
    return Rd.Clamp();
}
```

16.5.4 RENDERING WITH HIERARCHICAL INTEGRATION

The `DipoleSubsurfaceIntegrator::Li()` method performs standard direct lighting calculations for objects that don't have BSSRDFs. For those that do, the following fragment gets the medium's scattering properties from the `BSSRDF` object. The σ_a variable corresponds to the absorption coefficient introduced in Section 11.1, and σ_s' and σ_t' are the reduced scattering coefficient and reduced extinction coefficient, respectively, from Secton 16.5.3.

⟨*Evaluate BSSRDF and possibly compute subsurface scattering*⟩ ≡
```
BSSRDF *bssrdf = isect.GetBSSRDF(ray, arena);
if (bssrdf && octree) {
    Spectrum sigma_a  = bssrdf->sigma_a();
    Spectrum sigmap_s = bssrdf->sigma_prime_s();
    Spectrum sigmap_t = sigmap_s + sigma_a;
    if (!sigmap_t.IsBlack()) {
        ⟨Use hierarchical integration to evaluate reflection from dipole model 910⟩
    }
}
```

The R_d function defined in Equation (16.6) computes the diffuse subsurface reflectance between two points on the boundary of a semi-infinite scattering medium. We will now show how this function can be used to efficiently evaluate the subsurface scattering equation with a hierarchical integration scheme.

We start with a BSSRDF that is based on the diffusion subsurface reflectance approximation R_d,

$$S(p_o, \omega_o, p_i, \omega_i) = \frac{1}{\pi} F_t(\eta_o, \omega_o) R_d(\|p_i - p_o\|) F_t(\eta_i, \omega_i).$$

The division by π converts diffuse reflectance to a BSSRDF (though one that is a uniform function of direction), similar to how a BRDF's reflectance can be converted into a Lambertian BRDF (Section 8.1.1). The two Fresnel terms in turn account for the transmission of light through the boundary of the medium at the entrance and exit.

BSSRDF 598
BSSRDF::sigma_a() 599
BSSRDF::sigma_prime_s() 599
Intersection::GetBSSRDF() 484
M_PI 1002
Spectrum 263
Spectrum::Exp() 265
Spectrum::IsBlack() 265
Spectrum::Sqrt() 265

If this approximation is substituted into the subsurface scattering equation (Equation (5.9)), we have

$$L_o(p_o, \omega_o) = \int_A \int_{\mathcal{H}^2(\mathbf{n})} \left(\frac{1}{\pi} F_t(\eta_o, \omega_o) R_d(\|p_i - p_o\|) F_t(\eta_i, \omega_i) \right) L_i(p_i, \omega_i) |\cos \theta_i| \, d\omega_i \, dA.$$

This can be immediately simplified a bit, with the constant and $F_t(\eta_o, \omega_o)$ terms moved out of the integral entirely, since they are neither functions of position nor incident direction ω_i. In a similar manner, we can move the R_d term into just the area integral, since it doesn't depend on incident direction:

$$L_o(p_o, \omega_o) = \frac{1}{\pi} F_t(\eta_o, \omega_o) \int_A R_d(\|p_i - p_o\|) \left[\int_{\mathcal{H}^2(\mathbf{n})} F_t(\eta_i, \omega_i) L_i(p_i, \omega_i) |\cos \theta_i| \, d\omega_i \right] dA.$$

Note that, if not for the Fresnel transmittance term $F_t(\eta_i, \omega_i)$, the integral over incident directions ω_i in brackets would be the irradiance at the point p_i (Equation (5.3)). Therefore, the next approximation is to approximate this integral by the product of the irradiance at the point and the diffuse Fresnel transmittance $F_{dt}(\eta_i)$,

$$\int_{\mathcal{H}^2(\mathbf{n})} F_t(\eta_i, \omega_i) \, L_i(p_i, \omega_i) \, |\cos \theta_i| \, d\omega_i \approx F_{dt}(\eta_i) \, E(p_i, \mathbf{n}).$$

For a homogeneous medium, as we're assuming here, η is a constant, so $\eta = \eta_o = \eta_i$. We can apply the irradiance approximation above and move the F_{dt} term out of the integral over area, giving

$$L_o(p_o, \omega_o) \approx \frac{1}{\pi} F_t(\eta, \omega_o) \, F_{dt}(\eta) \int_A R_d(\|p_i - p_o\|) \, E(p_i, \mathbf{n}) \, dA.$$

The remaining terms in the integral over area give the radiant exitance M_o (introduced in Section 5.4) at the point p_o due to subsurface scattering from irradiance at the surface.

$$M_o(p_o) = \int_A R_d(\|p_i - p_o\|) \, E(p_i, \mathbf{n}) \, dA$$

We can approximate this integral as a sum over j area elements, each centered at a point p_j with area A_j and average irradiance E_j, giving

$$M_o(p_o) \approx \sum_j R_d(\|p_j - p_o\|) \, E_j \, A_j. \quad [16.7]$$

Intuitively, the error from this approximation depends on the variation of the respective terms; for example, if the R_d term actually varies significantly over the area of an element, then only evaluating it at the single point p_j may introduce unacceptable error. See Exercises 16.15 and 16.16 for further discussion of this issue and two ways of addressing it.

With this approach to computing M_o, we have the expression for computing outgoing radiance due to subsurface scattering that is used in the implementation below.

$$L_o(p_o, \omega_o) \approx \frac{1}{\pi} F_t(\omega_o)\, F_{dt}(p_o)\, M_o(p_o). \quad [16.8]$$

The terms other than $M_o(p_o)$ are computed trivially, and $M_o(p_o)$ is computed using the approximation of Equation (16.7), which is evaluated with the `SubsurfaceOctreeNode::Mo()` method.

⟨*Use hierarchical integration to evaluate reflection from dipole model*⟩ ≡ 908

```
DiffusionReflectance Rd(sigma_a, sigmap_s, bssrdf->eta());
Spectrum Mo = octree->Mo(octreeBounds, p, Rd, maxError);
FresnelDielectric fresnel(1.f, bssrdf->eta());
Spectrum Ft = Spectrum(1.f) - fresnel.Evaluate(AbsDot(wo, n));
float Fdt = 1.f - Fdr(bssrdf->eta());
L += (INV_PI * Ft) * (Fdt * Mo);
```

The `Mo()` method is called first with the root node of the tree. This method returns the radiant exitance at the point `pt` due to all of the `IrradiancePoints` in the nodes beneath it. At each node, it uses the aggregate values for the points beneath it, computed earlier in `InitHierarchy()`, if the error from using the clustered values would be low. Otherwise, it either sums the M_o values returned by its children nodes, or, if at a leaf, sums the contributions from each of the `IrradiancePoints` the leaf stores.

Note that there is an inaccuracy lurking in this approach: two separate objects that exhibit subsurface scattering end up using irradiance samples from the surface of the other object. In some cases, this is reasonable—consider a marble chess board made out of two types of marble in a checkered pattern where each check is modeled as an individual shape. The irradiance arriving at one check is reasonable to use for computing radiant exitance from its neighbor. On the other hand, the case illustrated in Figure 16.17, where the two objects are separated by a wall, is one where it's completely incorrect to use the other object's irradiance values and a per-primitive octree would give a better result. In general, the fall-off of the R_d function generally helps mitigate the visual impact of this issue.

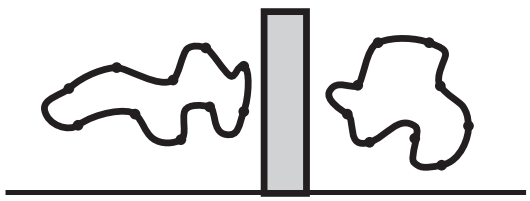


Figure 16.17: Setting That the Octree of Irradiance Points Doesn't Handle Well. With a single octree that holds all of the irradiance points in the scene, when we are computing radiant exitance from a point, we may inadvertently use irradiance samples from different objects. Here, the two objects are not only disconnected but also separated by a wall. The irradiance values on one's surface shouldn't affect the radiant exitance from the other, but the current implementation doesn't handle this case correctly.

BSSRDF::eta() 599
DiffusionReflectance 907
DipoleSubsurfaceIntegrator::octree 898
Fdr() 905
FresnelDielectric 437
FresnelDielectric::Evaluate() 437
INV_PI 1002
IrradiancePoint 898
Spectrum 263
SubsurfaceOctreeNode::InitHierarchy() 901
SubsurfaceOctreeNode::Mo() 911

⟨*DipoleSubsurfaceIntegrator Method Definitions*⟩ +≡
```
Spectrum SubsurfaceOctreeNode::Mo(const BBox &nodeBound, const Point &pt,
        const DiffusionReflectance &Rd, float maxError) {
    ⟨Compute Mo at node if error is low enough 911⟩
    ⟨Otherwise compute Mo from points in leaf or recursively visit children 911⟩
}
```

Two criteria are used to determine whether the clustered values at the node can be used. First, if the point being shaded is inside the node, then the clustered values are never used but the children are always visited (or the `IrradiancePoints` themselves are used, if in a leaf). The second criterion computes a conservative bound on the solid angle subtended by the area element as seen from the lookup point. If this value is less than a user-defined error bound and the lookup point is outside the node, then the clustered values at the node are used and recursion for the subtree beneath the node is skipped.

⟨*Compute M_o at node if error is low enough*⟩ ≡ 911
```
float dw = sumArea / DistanceSquared(pt, p);
if (dw < maxError && !nodeBound.Inside(pt))
    return Rd(DistanceSquared(pt, p)) * E * sumArea;
```

If the current node couldn't be used directly, then a finer approximation is used instead.

⟨*Otherwise compute M_o from points in leaf or recursively visit children*⟩ ≡ 911
```
Spectrum Mo = 0.f;
if (isLeaf) {
    ⟨Accumulate Mo from leaf node 911⟩
}
else {
    ⟨Recursively visit children nodes to compute Mo 911⟩
}
return Mo;
```

At leaf nodes, the best that can be done is to loop over the `IrradiancePoints` and apply Equation (16.7).

⟨*Accumulate M_o from leaf node*⟩ ≡ 911
```
for (int i = 0; i < 8; ++i) {
    if (!ips[i]) break;
    Mo += Rd(DistanceSquared(pt, ips[i]->p)) * ips[i]->E * ips[i]->area;
}
```

For interior nodes, the contributions of the non-`NULL` children are summed similarly.

⟨*Recursively visit children nodes to compute M_o*⟩ ≡ 911
```
Point pMid = .5f * nodeBound.pMin + .5f * nodeBound.pMax;
for (int child = 0; child < 8; ++child) {
    if (!children[child]) continue;
    BBox childBound = octreeChildBound(child, nodeBound, pMid);
    Mo += children[child]->Mo(childBound, pt, Rd, maxError);
}
```

BBox 70
BBox::Inside() 72
BBox::pMax 71
BBox::pMin 71
DiffusionReflectance 907
DistanceSquared() 65
IrradiancePoint 898
IrradiancePoint::area 898
IrradiancePoint::E 898
IrradiancePoint::p 898
octreeChildBound() 1026
Point 63
Spectrum 263
SubsurfaceOctreeNode::children 899
SubsurfaceOctreeNode::E 899
SubsurfaceOctreeNode::ips 899
SubsurfaceOctreeNode::isLeaf 899
SubsurfaceOctreeNode::Mo() 911
SubsurfaceOctreeNode::p 899
SubsurfaceOctreeNode::sumArea 899

16.5.5 SETTING SCATTERING PROPERTIES

It is remarkably unintuitive to set values of the absorption coefficient σ_a and the modified scattering coefficient σ_s' to achieve a desired visual result. If measured values of these parameters aren't available (for example, from the values made available from the `GetVolumeScatteringProperties()` utility function of Section 11.6.1), then the task for an artist trying to render subsurface scattering can be difficult.

Jensen and Buhler (2002) developed a different approach that derives scattering properties from a diffuse reflection color, the average distance light travels in the medium before scattering (the mean path length), and the index of refraction—much more intuitive parameters. The medium can be made more transparent by increasing the mean path length or more dense by decreasing it. The amount of multiple scattering can be controlled by how close the diffuse reflection color is to one.

The BRDF reflectance corresponding to a BSSRDF can be found by computing the integral of R_d to infinity; this gives the reflectance that results from a uniformly illuminated infinite surface with subsurface scattering. There is a compact closed form expression for this integral,

$$\int_0^\infty R_d(x)\,dx = \frac{\alpha'}{2}\left(1 + e^{-\frac{4}{3}A\sqrt{3(1-\alpha')}}\right) e^{-\sqrt{3(1-\alpha')}}, \tag{16.9}$$

with $\alpha' = \sigma_s'/\sigma_t'$ and A defined as in Section 16.5.3.

The `RdIntegral()` function returns the value of the integral for given α' and A values.

⟨*Volume Scattering Local Definitions*⟩ ≡
```
static float RdIntegral(float alphap, float A) {
    float sqrtTerm = sqrtf(3.f * (1.f - alphap));
    return alphap / 2.f * (1.f + expf(-4.f/3.f * A * sqrtTerm)) *
        expf(-sqrtTerm);
}
```

If we consider for now a single wavelength of light with given reflectance and A value, we'd like to find the α' value such that the integral of Equation (16.9) returns the same reflectance. The value of the integral is monotonic and smooth so that a binary search quickly converges to an accurate result.

⟨*Volume Scattering Local Definitions*⟩ +≡
```
static float RdToAlphap(float reflectance, float A) {
    float alphaLow = 0., alphaHigh = 1.f;
    float kd0 = RdIntegral(alphaLow, A);
    float kd1 = RdIntegral(alphaHigh, A);
    for (int i = 0; i < 16; ++i) {
        float alphaMid = (alphaLow + alphaHigh) * 0.5f;
        float kd = RdIntegral(alphaMid, A);
        if (kd < reflectance) { alphaLow = alphaMid;  kd0 = kd; }
        else                  { alphaHigh = alphaMid; kd1 = kd; }
    }
    return (alphaLow + alphaHigh) * 0.5f;
}
```

GetVolumeScattering Properties() 600
RdIntegral() 912

We can put this all together into a small utility function that takes given diffuse reflectance, mean path length, and index of refraction values and returns σ_a and σ_s'. It works one component at a time, after converting to RGB.

⟨*Volume Scattering Definitions*⟩ +≡
```
void SubsurfaceFromDiffuse(const Spectrum &Kd, float meanPathLength,
        float eta, Spectrum *sigma_a, Spectrum *sigma_prime_s) {
    float A = (1.f + Fdr(eta)) / (1.f - Fdr(eta));
    float rgb[3];
    Kd.ToRGB(rgb);
    float sigma_prime_s_rgb[3], sigma_a_rgb[3];
    for (int i = 0; i < 3; ++i) {
        ⟨Compute α′ for RGB component, compute scattering properties 913⟩
    }
    *sigma_a = Spectrum::FromRGB(sigma_a_rgb);
    *sigma_prime_s = Spectrum::FromRGB(sigma_prime_s_rgb);
}
```

Given the modified albedo α' that corresponds to the reflectance value given, the rest of the scattering properties can be computed. The inverse mean path length is approximately the effective transport coefficient σ_{tr} and we can use the relationship between σ_{tr}, σ_t', and α' from Section 16.5.3 to solve for σ_t'. Finally, the relationships $\sigma_s' = \alpha'\sigma_t'$ and $\sigma_a = \sigma_t' - \sigma_s'$ are used to compute σ_s' and σ_a.

⟨*Compute α′ for RGB component, compute scattering properties*⟩ ≡ 913
```
float alphap = RdToAlphap(rgb[i], A);
float sigma_tr = 1.f / meanPathLength;
float sigma_prime_t = sigma_tr / sqrtf(3.f * 1.f - alphap);
sigma_prime_s_rgb[i] = alphap * sigma_prime_t;
sigma_a_rgb[i] = sigma_prime_t - sigma_prime_s_rgb[i];
```

FURTHER READING

Lommel (1889) was apparently the first to derive the equation of transfer, in a not widely known paper. Not only did he derive the equation of transfer, but he also solved it in some simplified cases in order to estimate reflection functions from real-world surfaces (including marble and paper) and compared his solutions to measured reflectance data from these surfaces.

Seemingly unaware of Lommel's work, Schuster (1905) was the next researcher in radiative transfer to consider the effect of multiple scattering. He used the term *self-illumination* to describe the fact that each part of the medium is illuminated by every other part of the medium, and he derived differential equations that described reflection from a slab along the normal direction assuming the presence of isotropic scattering. The conceptual framework that he developed remains essentially unchanged in the field of radiative transfer.

Soon thereafter, Schwarzschild (1906) introduced the concept of radiative equilibrium, and Jackson (1910) expressed Schuster's equation in integral form, also noting that

Fdr() 905
RdToAlphap() 912
Spectrum 263
Spectrum::FromRGB() 277
Spectrum::ToRGB() 275

"the obvious physical mode of solution is Liouville's method of successive substitutions" (i.e., a Neumann series solution). Finally, King (1913) completed the rediscovery of the equation of transfer by expressing it in the general integral form. Yanovitskij (1997) traced the origin of the integral equation of transfer to Chvolson (1890), but we have been unable to find a copy of this paper.

Blinn (1982b) first used basic volume scattering algorithms for computer graphics. The equation of transfer was first introduced to graphics by Kajiya and Von Herzen (1984). Rushmeier (1988) was the first to compute solutions of it in a general setting. Arvo (1993) first made the essential connections between previous formalizations of light transport in graphics and the equation of transfer and radiative transfer in general.

Other early work in volume scattering for computer graphics includes work by Max (1986) and Nishita, Miyawaki, and Nakamae (1987). Glassner (1995) provided a thorough overview of this topic and previous applications of it in graphics, and Max's survey article (Max 1995) also covers the topic well. One key application of volume scattering algorithms in computer graphics has been simulating atmospheric scattering. Work in this area includes papers by Klassen (1987) and Preetham, Shirley, and Smits (1999), who introduced a physically rigorous and computationally efficient atmospheric and skylighting model.

Researchers have recently had success in deriving closed-form expressions that describe scattering in participating media under certain assumptions; these approaches can be substantially more efficient than previous methods. See Sun et al. (2005), Pegoraro and Parker (2009), and Pegoraro et al. (2009) for examples of such methods. (Remarkably, Pegoraro and collaborators' work provides a closed-form expression for scattering from a point light source along a ray passing through homogeneous participating media with anisotropic phase functions.)

Rushmeier and Torrance (1987) used finite-element methods for rendering participating media, and Lafortune and Willems (1996) applied bidirectional path tracing to the problem. Other work includes Bhate and Tokuta's approach based on spherical harmonics (Bhate and Tokuta 1992) and Blasi et al.'s two-pass Monte Carlo algorithm, where the first pass shoots energy from the lights and stores it in a grid and the second pass does final rendering using the grid to estimate illumination at points in the scene (Blasi, Saëc, and Schlick 1993). More recently, Szirmay-Kalos et al. (2005) precomputed interacitions between sample points in the medium in order to more quickly compute multiple scattering. See Cerezo et al. (2005) for an extensive survey of approaches to rendering participating media.

Jensen and Christensen (1998) generalized the photon mapping algorithm for rendering participating media, and Jarosz et al. (2008) showed how to express the scattering integral over a beam through the medium as a measurement to be evalauted with particles from a particle-tracing algorithm (a similar measurement-based approach was introduced for reflection from surfaces in Section 15.6.1). This led to a much more efficient implementation of volume scattering. Pauly, Kollig, and Keller (2000) generalized the Metropolis light transport algorithm to include volume scattering. Pauly's thesis (Pauly 1999) described the theory and implementation of bidirectional and Metropolis-based algorithms for volume light transport. See Pegoraro et al. (2008b) for an interesting approach for

improving Monte Carlo rendering of participating media by using information from previous samples to guide future sampling.

Moon et al. (2007) made the important observation that some of the assumptions underlying the use of the equation of transfer—that the scattering particles in the medium aren't too close together so that scattering events can be considered to be statistically independent—aren't in fact true for interesting scenes that include small crystals, ice, or piles of many small glass objects. They developed a new light transport algorithm for these types of *discrete random media* based on composing precomputed scattering solutions.

There are a number of applications of visualizing volumetric data sets for medical and engineering applications. This area is called *volume rendering*. In many of these applications, radiometric accuracy is substantially less important than developing techniques that help make structure in the data apparent (e.g., where the bones are in CT scan data). Early papers in this area include those by Levoy (1988, 1990a, 1990b) and Drebin, Carpenter, and Hanrahan (1988).

Subsurface scattering was first introduced to graphics by Hanrahan and Krueger (1993), although their approach did not attempt to simulate light that entered the object at points other than at the point being shaded. Dorsey et al. (1999) applied photon maps to simulating subsurface scattering that did include this effect, and Pharr and Hanrahan (2000) introduced an approach based on computing BSSRDFs for arbitrary scattering media by integrating over the medium.

Kajiya and Von Herzen (1984) first introduced the diffusion approximation to graphics, though Stam (1995) was the first to clearly identify many of its advantages for rendering. See Ishimaru's book (1978) or Donner's thesis (2006) for the derivation of the diffusion approximation and Wyman et al. (1989) for the introduction of the principle of similarity.

The dipole approximation for subsurface scattering was developed by Farrell et al. (1992). It was introduced to computer graphics by Jensen et al. (2001b). Jensen and Buhler (2002) developed the hierarchical integration approach implemented here in the `DipoleSubsurfaceIntegrator`. The dipole approximation has since been the basis of a number of fast implementations for scan-line and interactive rendering (Hery 2003; Hao, Baby, and Varshney 2003; Dachsbacher and Stamminger 2003). Arbree et al. (2008) observed that the two-pass hierarchical integration approach doesn't scale well to complex scenes: excessive time may be spent generating irradiance samples in parts of the scene that aren't needed for rendering the final image. They proposed a one-pass approach based on the lightcuts method that generates irradiance samples on demand, clustering them for efficient integration. (See the "Further Reading" section in Chapter 15 for further information on lightcuts.)

Contini et al. (1997) generalized the dipole approach to *multipoles* to more accurately model finite scattering slabs. This approach was applied to subsurface scattering by Donner and Jensen (2005). However, even the multipole approach doesn't handle all types of scattering media well; the asssumptions of heterogeneous media and relatively high albedos are too restrictive for many interesting objects. Li et al. (2005) developed a hybrid approach that handles the first few bounces of light with Monte Carlo path tracing

DipoleSubsurfaceIntegrator 887

but then switches to a dipole approximation. Tong et al. (2005), Haber et al. (2005), and Wang et al. (2008b) further generalized the media supported, solving the diffusion equation on a grid of sample points. Fattal (2009) applied the discrete ordinates method, addressing a number of shortcomings of the direct application of that technique.

Donner et al. (2009) computed BSSRDFs with Monte Carlo simulation for a variety of scattering properties (phase function, scattering coefficients, etc.), and fit the resulting data to a low-dimensional model. This model much better accounts for the directional variation of scattered light and the properties of medium-albedo media.

Rendering realistic human skin is a challenging problem; this problem has driven the development of a number of new methods for rendering subsurface scattering after the initial dipole work as issues of modeling the layers of skin and computing more accurate simulations of scattering between layers have been addressed. For a good overview of these issues, see Igarashi et al.'s (2007) survey on the scattering mechanisms inside skin and approaches for measuring and rendering skin. Notable research in this area includes papers by Donner and Jensen (2006), d'Eon et al. (2007), Ghosh et al. (2008), and Donner et al. (2008). Donner's thesis includes a discussion of the importance of accurate spectral representations for high-quality skin rendering (Donner 2006, Section 8.5).

The Poisson sphere point distribution technique in the `DipoleSubsurfaceIntegrator` is based on the approach described by Lehtinen et al. (2008). Arbree et al. (2008) used a similar approach to distribute points for subsurface scattering. Bikker and Reijerse (2009) developed a related technique that tries to adapt the point distribution, placing more samples in areas where the illumination is changing rapidly. A much more efficient approach for placing Poisson disk-distributed points on surfaces was described by Cline et al. (2009a), though they require that a number of operations can be performed for each type of shape used. Their paper also has a good survey of previous work in this area.

EXERCISES

❷ 16.1 With inhomogeneous volume regions, where the optical depth between two points must be computed with ray marching, the `SingleScatteringIntegrator` volume integrator may spend a lot of time finding the attenuation between lights and points on rays where single scattering is being computed. One approach to reducing this computation is to take advantage of the fact that the amount of attenuation for nearby rays is generally smoothly varying and to use a precomputed approximation to the attenuation. For example, Kajiya and Von Herzen (1984) computed the attenuation to a directional light source at a grid of points in 3D space and then found attenuation at any particular point by interpolating among nearby grid samples. A more memory-efficient approach was developed by Lokovic and Veach (2000) in the form of deep shadow maps, based on a clever compression technique that takes advantage of the smoothness of the attenuation. Implement one of these approaches in `pbrt` and measure how much it speeds up the `SingleScatteringIntegrator` integrator. Under what sort of situations do approaches like these result in image errors?

DipoleSubsurfaceIntegrator 887
SingleScatteringIntegrator 882

● 16.2 Another effective method to speed up the `SingleScatteringIntegrator` is to eliminate the need to trace shadow rays from points along the ray passed to the `SingleScatteringIntegrator::Li()` method by computing shadow maps from each light source (Williams 1978; Reeves, Salesin, and Cook 1987) and use them to determine light source visibility in place of shadow rays. Modify `pbrt` to optionally use this approach, and measure the performance difference. What resolution do you find to be necessary in the shadow maps for high-quality results?

● 16.3 One shortcoming of the `VolumeIntegrators` in this chapter is that they always take a fixed step size along the ray through the participating medium. If the medium is very dense in some parts but sparse in others, this may be inefficient, as a short step size is needed to accurately resolve detail in the thick parts but is wasteful in the rest of the volume. Another way that this integration can be performed is to sample a beam transmittance at which to compute a scattering event, find the point along the ray where it has passed through that thickness, and then do the scattering computation there. For constant-density volumes, the corresponding offset along the ray can be found in closed form. For varying density volumes, ray marching will still in general be required, though with less overhead than the current implementation, since a full lighting calculation isn't being performed at each ray-marching point.

Add a method to the `VolumeRegion` class that supports sampling a given optical thickness along a ray and implement it for the various implementations. Derive the closed-form method to sample the distance at which a given beam transmittance value is reached for a homogeneous medium, and implement a ray-marching algorithm that computes this for general volumes. Then, modify the `SingleScatteringIntegrator` to use it. What is the performance difference when using your new approach, for rendering images of approximately equal quality as the original implementation?

● 16.4 Performing ray-marching to sample the point at which a desired beam transmittance value is reached as in Exercise 16.3 can still be computationally expensive. Coleman (1968) shows that if the maximum scattering coefficient of the medium is known, then an unbiased approach is to generate a candidate sample as if the medium was homogeneous, with a scattering coefficient equal to the maximum value, and then to probabilistically accept the candidate point and compute scattering with probability equal to the ratio of the actual scattering coefficient at the point and the maximum scattering coefficient. (This approach was introduced to graphics by Matthias Rabb (Shirley 2009).) Implement this technique and evaluate its effectiveness.

● 16.5 An additional advantage of the approaches in Exercises 16.3 and 16.4 compared to the emission-only and single scattering integrator implementations in this chapter is that the points at which scattering is computed end up being exponentially distributed—they are more likely to be close to the start of the ray, where less attenuatation has occurred and thus the contributions of the samples taken will have been reduced less by the beam transmittance term. However,

SingleScatteringIntegrator 882
SingleScatteringIntegrator::Li() 882
VolumeIntegrator 876
VolumeRegion 587

there are cases where this isn't the optimal approach, either. Consider a medium where there is no illumination in the front, as seen by the camera, but an extremely bright light is shining in the back part of it. Even though the points in the back will be attenuated more, they're the only ones where the scattering term will be nonzero!

The general problem is that we need to compute a Monte Carlo estimate of a function that's comprised of the product of two functions: the transmittance and the source term. Each of them should be able to direct where sampling happens, with the goal that the combined function is sampled at points where the product of the two functions is large. The general problem here is similar to the case of sampling direct illumination at a surface from a small light source: sampling the BSDF is an inefficient approach, but sampling using the light works quite well. In some cases, sampling the ray is inefficient, but sampling using the light is effective. (As another example, consider a spotlight with a small cone angle shining light through a participating medium. The `SingleScatteringIntegrator` integrator will find it very difficult to efficiently render an image that accurately captures the beam, since a small step size will be needed to find points along rays that lie in the spotlight's cone, yet a small step size will be inefficient for most of the rays that don't pass through the cone at all (Nishita, Miyawaki, and Nakamae 1987).)

Modify `pbrt`'s light sampling interfaces and the `SingleScatteringIntegrator` implementation so that light sources are able to convey information about which parts of particular rays they potentially illuminate. One possibility would be to allow light sources to return a parametric t range to sample for each ray; a default implementation could set the range to $[0, \infty)$. Another possible approach would be to allow the lights the opportunity to sample points along rays directly. Implement one of these techniques in `pbrt` and measure the improvement in running time for difficult-to-render scenes like the one described earlier.

❷ 16.6 Extend the `SingleScatteringIntegrator` to use multiple importance sampling based on sampling points on the light source and sampling the phase function for the direct lighting computation. Under what circumstances will this method give substantially less variance than the current implementation?

❸ 16.7 Design and implement a general Monte Carlo path-tracing approach to compute images with multiple scattering in participating media. (To combat the high running times of these algorithms, it may be worthwhile to first implement one of the techniques for reducing the cost of shadow rays described in earlier exercises.) For background information and general approaches, it may be helpful to read Lafortune and Willems's paper about bidirectional path tracing in participating media, even if you don't implement a bidirectional algorithm (Lafortune and Willems 1996). Furthermore, Pauly, Kollig, and Keller (2000) derived the generalization of the path integral form of the light transport equation for the volume scattering case; their insights may be useful as well. Render images that show the effect of multiple scattering in participating media.

SingleScatteringIntegrator 882

❸ 16.8 Photon mapping has also been shown to be an effective way to model the effect of multiple scattering in participating media. Read Jensen and Christensen's paper on this topic (Jensen and Christensen 1998) and implement this method in a new `VolumeIntegrator` in `pbrt`. You may find that the improved method for sampling and interpolating the photons described by Jarosz et al. (2008) substantially improves the efficiency of your implementation.

❸ 16.9 Ren et al. (2008) described an approach for efficiently rendering participating media where both radiance and its gradient are computed on an adaptive grid and then radiance values are interpolated with these two quantities. Read their paper and implement this approach as a `VolumeIntegrator` in `pbrt`. How does efficiency compare to the `SingleScatteringIntegrator`?

❷ 16.10 Recall from Section 16.5.3 that the diffusion approximation doesn't include the effect of direct transmission through the medium and light that is scattered once in the medium. Modify the `DipoleSubsurfaceIntegrator::Li()` method so that it accounts for these two currently missing terms. For direct transmission, you will want to compute the refracted ray that enters the object and find the intersection point where it leaves the object. The incident radiance along the ray that refracts back out of the object should then be attenuated by the transmittance of the ray inside the object. (It may be worthwhile to use Russian roulette to avoid computing incident radiance along the exiting ray for cases where the transmittance is very low.)

For single scattering, you will want to sample one or more points along the segment inside the object and compute single scattering along the lines of the `SingleScatteringIntegrator`. There are two subtleties with an implementation in this context. First, it's important to sample scattering from an exponential distribution along the segment: because the attenuation coefficient is relatively large in media with high albedos, single scattering at points that aren't close to the ray's entry-point to the medium makes a low contribution to the final result. Second, you will need to ignore intersections with the object's surface for shadow rays traced from the scattering point. (One way to do this is to trace initial shadow rays until they intersect the surface and then trace a second ray to do the actual shadow occlusion computation.)

After implementing these changes, compare images rendered with and without these two terms. How much of a difference do they make in practice? For what sorts of models is the difference most evident?

❸ 16.11 Implement a subsurface scattering integrator that uses Monte Carlo path tracing to compute multiple scattering. Verify the correctness of your implementation by comparing results when rendering a semi-infinite slab—the case with the lowest expected error for the diffusion dipole approach. Recall that in high-albedo media, paths of hundreds or thousands of bounces may be necessary to compute accurate results. Render images that demonstrate cases where the dipole approximation introduces excessive error but Monte Carlo computes a correct result. How much slower is the Monte Carlo approach for cases where the dipole is accurate?

DipoleSubsurfaceIntegrator::Li() 908
SingleScatteringIntegrator 882
VolumeIntegrator 876

16.12 As discussed in Section 16.5.1, the random ray-tracing approach for placing Poisson-disk points on surfaces can be computationally expensive. Read the paper by Cline et al. (2009a) on a more efficient approach for placing Poisson disk points on surfaces and implement their approach. Modify the `DipoleSubsurfaceIntegrator` to loop over all primitives that have a BSSRDF (you'll need to add a method to the `Primitive` interface that reports this information), and modify the `Shape` interface as needed to implement their approach. How much faster is this approach than the one in the current implementation? Does it have any disadvantages?

16.13 Generalize the irradiance computation in the `DipoleSubsurfaceIntegrator` to create and use one of the other `Integrators` in `pbrt` so that indirect illumination is included in the computed irradiance values. As part of this, you will probably want to break the irradiance computation into tasks so that they can run in parallel. Render images that show the improvement from including indirect lighting with a test scene.

16.14 Approximations of the dipole model such that the medium is semi-infinite and homogeneous can lead to substantial errors when these conditions are not true. Read Donner and Jensen's paper (2005) on using multiple dipoles (multipoles) to render thin translucent materials and on new techniques to handle media with layers with differing scattering properties. Implement their method in `pbrt` and render images showing the improved results when rendering thin translucent objects.

16.15 With the `DipoleSubsurfaceIntegrator`'s hierarchical integration scheme, it can be necessary to generate irradiance samples at a high density on the surface to achieve good results. (Figure 16.18 shows the characteristic splotchy artifacts that result from an insufficient number of samples.) Langlands and Mertens (2007) observed that these artifacts are the result of the rapid falloff of the subsurface reflectance term R_d; when shading some points, there will be a nearby irradiance sample that will match up with the peak of R_d and make a large contribution, but at others there won't be any samples sufficiently close by. (Recall the graph of the R_d function in Figure 16.16.)

To address this problem, Langlands and Mertens proposed breaking the subsurface scattering computation into two terms: one for nearby points and one for the rest. The average irradiance, E_n, of the nearby points up to some distance d is computed; a weighting function of distance $w(x)$ blends between the contributions of near and far irradiance samples, and Equation (16.8) is generalized to:

$$L_o(p_o, \omega_o) \approx 2\pi E_n \int_0^d w(x)\, R_d(x)\, dx + \frac{1}{\pi} F_t(\omega_o)\, F_{dt}(p_o) \sum_j (1 - w(\|p_j - p_o\|)) R_d(\|p_j - p_o\|)\, E_j\, A_j.$$

Thus, farther away away points are handled using the usual approach. They suggest using a hat function $w(x) = \max(0, 1 - d)$ for the weighting function and

DipoleSubsurfaceIntegrator 887
Integrator 740
Primitive 185
Shape 108

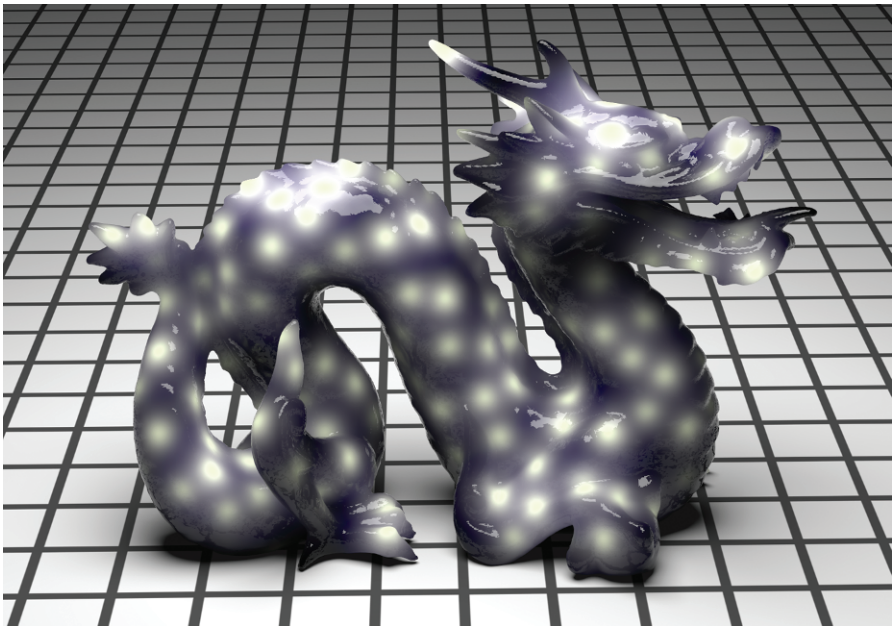


Figure 16.18: Characteristic Splotchy Artifacts from Sampling Irradiance at Too Low a Rate with the `DipoleSubsurfaceIntegrator`. The model from Figure 16.9 is rendered with not enough irradiance samples. The image has artifacts due to the rapid fall-off of the R_d function, such that there are no irradiance samples near its peak at many of the points being shaded. Exercises 16.15 and 16.16 outline solutions to this problem.

selecting the distance d so that it includes ten or so irradiance samples. For efficiency, the integral of the product of $w(x)$ and $R_d(x)$ should be precomputed and tabularized.

Implement this method in `pbrt`. How much lower an irradiance point sampling frequency can you use to get good results versus the current implementation?

❷ 16.16 Another approach for reducing the splotchy artifacts from the rapid R_d function fall-off was suggested by Neulander (2009). He introduced the idea of computing the incident irradiance at each outgoing point p_o where the hierarchical integration is being performed. Then, whichever point in the octree of irradiance points is closest to the outgoing point is replaced with a sample that has the same position and irradiance as the actual lookup point. This approach ensures that one sample is always taken at the peak of the subsurface reflectance function. Implement this approach for rendering subsurface scattering in `pbrt` and compare the results to the builtin dipole integrator.

❸ 16.17 Donner et al. (2009) performed extensive numerical simulation of subsurface scattering from media with a wide range of scattering properties and then computed coefficients to fit an analytical model to the resulting data. They have shown that rendering with this model is more efficient than full Monte Carlo integration, while handling well many cases where the approximations

DipoleSubsurfaceIntegrator 887

of the dipole model are unacceptable. For example, their model accounts for directional variation in the scattered radiance and handles media with low and medium albedos well. Read their paper and download the data files of coefficients. Implement a new subsurface integrator in `pbrt` that uses their model.

To implement this model, you will need a way to generate sample points on the surface around the point being shaded. One way to do this is to precompute a large number of points on the surface (along the lines of the `SurfacePointsRenderer`) and then store them in a spatial data structure like a kd-tree so that you can efficiently find points around a lookup point.

3 16.18 Modify `pbrt`'s interfaces and abstractions so that fully varying 3D scattering properties can be used for subsurface scattering. You may want to base your implementation on a grid of sample values, with the boundary defined by a `Primitive`'s surface. Then, implement a more general subsurface light transport simulation algorithm that handles 3D variation such as the one described by Tong et al. (2005), Haber et al. (2005), Wang et al. (2008b), or Fattal (2009).

SurfacePointsRenderer 888