CHAPTER TWELVE

12 LIGHT SOURCES

In order for objects in a scene to be visible, some of them must emit light that is eventually reflected back to the camera. This chapter introduces the abstract `Light` class, which defines the interface used for light sources in `pbrt`, as well as the implementations of a number of useful light sources. By hiding the implementation of different types of lights behind a carefully designed interface, the light transport routines can operate without knowing which particular types of lights are in the scene, similar to how the acceleration structures can hold collections of primitives without needing to know their actual types. This chapter only defines the basic light functionality because many of the quantities related to complex light sources cannot be computed in closed form. The Monte Carlo routines in Chapter 14 will complete the lighting interface and implementations by introducing methods for numerically approximating these quantities.

A range of light source implementations are introduced in this chapter, although the variety is slightly limited by `pbrt`'s physically based design. Many flexible light source models have been developed for computer graphics, incorporating control over properties like the rate at which the light falls off with distance, which objects are illuminated by the light, which objects cast shadows from the light, and so on. While controls such as these are quite useful for artistic effects, many of them are incompatible with physically based light transport algorithms and thus can't be provided in the models here. As an example of this issue, consider a light that doesn't cast shadows: the total energy arriving at surfaces in the scene increases without bound as more surfaces are added. Consider a series of concentric shells of spheres around such a light; if occlusion is ignored, each added shell increases the total received power. This directly violates the principle that the total energy arriving at surfaces illuminated by the light can't be greater than the total energy emitted by the light.

12.1 LIGHT INTERFACE

The core lighting routines and interfaces are in `core/light.h` and `core/light.cpp`. Implementations of particular lights are in individual source files in the `lights/` directory. All lights share two common parameters: a transformation that defines the light's coordinate system in world space and a parameter that affects Monte Carlo sampling of lights, `nSamples`. As with shapes, it's often handy to be able to implement a light assuming a particular coordinate system (e.g., that a spotlight is always located at the origin of its light space, shining down the $+z$ axis). The light-to-world transformation makes it possible to place such lights at arbitrary positions and orientations in the scene. The `nSamples` parameter is used for area light sources where it may be desirable to trace multiple shadow rays to the light to compute soft shadows; it allows the user to have finer-grained control of the number of samples taken on a per-light basis.

⟨*Light Declarations*⟩ ≡
```
class Light {
public:
    ⟨Light Interface 606⟩
    ⟨Light Public Data 606⟩
protected:
    ⟨Light Protected Data 606⟩
};
```

Although storing both the light-to-world and the world-to-light transformations is redundant, having both available simplifies code elsewhere by eliminating the need for calls for `Inverse()`. The default number of light source samples taken is one; thus, only the light implementations for which taking multiple samples is sensible need to pass in an explicit value here. The only other job for the constructor is to warn if the light-to-world transformation has a scale factor; many of the `Light` methods will return incorrect results in this case.[1]

⟨*Light Interface*⟩ ≡ 606
```
Light(const Transform &l2w, int ns = 1)
    : nSamples(max(1, ns)), LightToWorld(l2w),
      WorldToLight(Inverse(l2w)) {
    ⟨Warn if light has transformation with scale⟩
}
```

⟨*Light Protected Data*⟩ ≡ 606
```
const Transform LightToWorld, WorldToLight;
```

⟨*Light Public Data*⟩ ≡ 606
```
const int nSamples;
```

`Inverse()` 1021
`Light` 606
`Light::LightToWorld` 606
`Light::nSamples` 606
`Light::WorldToLight` 606
`Transform` 76

1 For example, the surface area reported by area lights is computed from the untransformed geometry, so a scale factor in the transformation means that the reported area and the actual area of the light in the scene would be inconsistent.

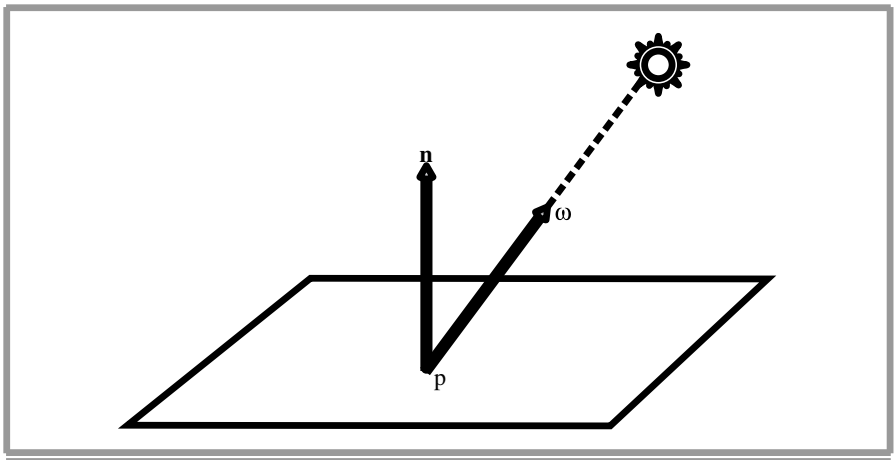


Figure 12.1: The `Light::Sample_L()` method returns incident radiance from the light at a point p and also returns the direction vector ω that gives the direction from which radiance is arriving.

The key method for lights to implement is `Sample_L()`. The caller passes the world space position of a point in the scene and the time at which the light sample is to be taken, and the light returns the radiance arriving at that point due to that light, assuming there are no occluding objects between them (Figure 12.1). Although this method also takes a time value, the `Light` implementations do not currently support being animated themselves—the lights themselves are at fixed positions in the scene. (Addressing this limitation is left as an exercise.) However, the time value is needed to set the corresponding value appropriately in the traced visibility ray.

The light is also responsible for initializing the incident direction to the light source ω_i and for initializing the `VisibilityTester` object, which holds information about the shadow ray that must be traced to verify that there are no occluding objects between the light and p. The `VisibilityTester`, which will be described in Section 12.1.1, need not be initialized if the returned radiance value is black—for example, due to the point p being outside of the cone of illumination of a spotlight. In addition to the point p being illuminated, this method takes an epsilon value for shadow rays leaving p, `pEpsilon`.

For some types of lights, light may arrive at p from many directions, not just from a single direction as with a point light source, for example. For these types of light sources, the `Sample_L()` method must randomly sample a point on the light source's surface, so that Monte Carlo integration can be used to find the reflected light at p due to illumination from the light. Implementations of the `Sample_L()` interface for such lights will be introduced later, in Section 14.6. The `LightSample` is used by these methods, and the `pdf` output parameter stores the probability density for the light sample taken. For all of the implementations in this chapter, the `LightSample` is ignored and the `pdf` is set to one. The `pdf` value's role in the context of Monte Carlo sampling is discussed in Section 14.6.1.

`Light` 606
`Light::Sample_L()` 608
`LightSample` 710
`VisibilityTester` 608

⟨*Light Interface*⟩ +≡ 606
```
virtual Spectrum Sample_L(const Point &p, float pEpsilon,
    const LightSample &ls, float time, Vector *wi, float *pdf,
    VisibilityTester *vis) const = 0;
```

All lights must also be able to return their total emitted power; this quantity is useful for light transport algorithms that may want to devote additional computational resources to lights in the scene that make the largest contribution. Because a precise value for emitted power isn't needed elsewhere in the system, a number of the implementations of this method later in this chapter will compute approximations to this value rather than expending computational effort to find a precise value.

⟨*Light Interface*⟩ +≡ 606
```
virtual Spectrum Power(const Scene *) const = 0;
```

Finally, the `IsDeltaLight()` method indicates whether the light is described by a delta distribution. Examples of such lights include point lights, which emit illumination from a single point, and directional lights, where all light arrives from the same direction. The only way to detect illumination from light sources like these is to call their `Sample_L()` methods. It's impossible to randomly choose a direction from a point p that happens to find such a light source. (This is analogous to delta components in BSDFs from specular reflection or transmission.) The Monte Carlo algorithms that sample illumination from light sources need to be aware of which lights are described by delta distributions, since this affects some of their computations.

⟨*Light Interface*⟩ +≡ 606
```
virtual bool IsDeltaLight() const = 0;
```

12.1.1 VISIBILITY TESTING

The `VisibilityTester` is a *closure*—an object that encapsulates a small amount of data and some computation that is yet to be done. It allows lights to return a radiance value under the assumption that the receiving point p and the light source are mutually visible. The integrator can then decide if illumination from the direction ω is relevant before incurring the cost of tracing the shadow ray—for example, light incident on the back side of a surface that isn't translucent contributes nothing to reflection from the other side. If the actual amount of arriving illumination is in fact needed, a call to one of the visibility tester's methods causes the necessary shadow ray to be traced.

⟨*Light Declarations*⟩ +≡
```
struct VisibilityTester {
    ⟨VisibilityTester Public Methods 609⟩
    Ray r;
};
```

There are two methods that initialize `VisibilityTesters`. The first of them, `VisibilityTester::SetSegment()`, indicates that the visibility test is to be done between two points in the scene. In addition to the two points, it takes "epsilon" values to offset the ray's starting point and ending point, respectively.

LightSample 710
Point 63
Ray 66
Scene 22
Spectrum 263
Vector 57
VisibilityTester 608
VisibilityTester::SetSegment() 609

⟨*VisibilityTester Public Methods*⟩ ≡ 608
```
void SetSegment(const Point &p1, float eps1,
                const Point &p2, float eps2, float time) {
    float dist = Distance(p1, p2);
    r = Ray(p1, (p2-p1) / dist, eps1, dist * (1.f - eps2), time);
}
```

The other initialization method, `VisibilityTester::SetRay()`, indicates that the test should indicate whether there is *any* object along a given direction. This variant is useful for computing shadows from directional lights.

⟨*VisibilityTester Public Methods*⟩ +≡ 608
```
void SetRay(const Point &p, float eps, const Vector &w, float time) {
    r = Ray(p, w, eps, INFINITY, time);
}
```

The next pair of methods trace the appropriate ray. The first one, `VisibilityTester::Unoccluded()` traces the shadow ray and returns a Boolean result. Some ray tracers include a facility for casting colored shadows from partially transparent objects and would return a spectrum from a method like this. `pbrt` does not include this facility explicitly, since those systems typically implement it with a nonphysical hack. Thus, the `VisibilityTester::Unoccluded()` method here returns a Boolean. Scenes where illumination passes through a transparent object should be rendered with an integrator that supports this kind of effect, like the `PhotonIntegrator` described in Section 15.6.

⟨*Light Method Definitions*⟩ ≡
```
bool VisibilityTester::Unoccluded(const Scene *scene) const {
    return !scene->IntersectP(r);
}
```

`VisibilityTester::Transmittance()` determines the fraction of illumination from the light to the point that is not extinguished by participating media in the scene. If the scene has no participating media, this method returns a constant spectral value of one.

⟨*Light Method Definitions*⟩ +≡
```
Spectrum VisibilityTester::Transmittance(const Scene *scene,
        const Renderer *renderer, const Sample *sample,
        RNG &rng, MemoryArena &arena) const {
    return renderer->Transmittance(scene, RayDifferential(r), sample,
                                   rng, arena);
}
```

INFINITY 1002
MemoryArena 1015
PhotonIntegrator 802
Point 63
PointLight 610
Ray 66
RayDifferential 69
Renderer 24
Renderer::Transmittance() 25
RNG 1003
Sample 343
Scene 22
Scene::IntersectP() 24
Spectrum 263
Vector 57
VisibilityTester 608
VisibilityTester::r 608
VisibilityTester::SetRay() 609
VisibilityTester::Unoccluded() 609

12.2 POINT LIGHTS

A number of interesting lights can be described in terms of emission from a single point in space with some possibly angularly varying distribution of outgoing light. This section describes the implementation of a number of them, starting with `PointLight`, which represents an isotropic point light source that emits the same amount of light in all directions. It is defined in `lights/point.h` and `lights/point.cpp`. Figure 12.2 shows

Figure 12.2: Scene Rendered with a Point Light Source. Notice the hard shadow boundaries from this type of light.

a scene rendered with a point light source. Building on this base, a number of more complex lights based on point sources will be introduced, including spotlights and a light that projects an image into the scene.

⟨*PointLight Declarations*⟩ ≡
```
class PointLight : public Light {
public:
    ⟨PointLight Public Methods 611⟩
private:
    ⟨PointLight Private Data 611⟩
};
```

PointLights are positioned at the origin in light space. To place them elsewhere, the light-to-world transformation should be modified as appropriate. Using this transformation, the world space position of the light is precomputed and cached in the constructor by transforming (0, 0, 0) from light space to world space. The constructor also stores the intensity for the light source, which is the amount of power per unit solid angle. Because the light source is isotropic, this is a constant.

⟨*PointLight Method Definitions*⟩ ≡
```
PointLight::PointLight(const Transform &light2world,
                       const Spectrum &intensity)
    : Light(light2world) {
    lightPos = LightToWorld(Point(0,0,0));
    Intensity = intensity;
}
```

Light 606
Light::LightToWorld 606
Point 63
PointLight 610
PointLight::Intensity 611
PointLight::lightPos 611
Spectrum 263
Transform 76

⟨*PointLight Private Data*⟩ ≡ 610
```
Point lightPos;
Spectrum Intensity;
```

Strictly speaking, it is incorrect to describe the light arriving at a point due to a point light source using units of radiance. Radiant intensity is instead the proper unit for describing emission from a point light source, as explained in Section 5.4. In the light source interfaces here, however, we will abuse terminology and use `Sample_L()` methods to report the illumination arriving at a point for all types of light sources, dividing radiant intensity by the squared distance to the point p to convert units. Section 14.6 revisits the details of this issue in its discussion of how delta distributions affect evaluation of the integral in the scattering equation. In the end, the correctness of the computation does not suffer from this fudge, and it makes the implementation of light transport algorithms more straightforward by not requiring them to use different interfaces for different types of light.

⟨*PointLight Method Definitions*⟩ +≡
```
Spectrum PointLight::Sample_L(const Point &p, float pEpsilon,
        const LightSample &ls, float time, Vector *wi, float *pdf,
        VisibilityTester *visibility) const {
    *wi = Normalize(lightPos - p);
    *pdf = 1.f;
    visibility->SetSegment(p, pEpsilon, lightPos, 0., time);
    return Intensity / DistanceSquared(lightPos, p);
}
```

The total power emitted by the light source can be found by integrating the intensity over the entire sphere of directions:

$$\Phi = \int_{S^2} I \, d\omega = I \int_{S^2} d\omega = 4\pi I.$$

⟨*PointLight Method Definitions*⟩ +≡
```
Spectrum PointLight::Power(const Scene *) const {
    return 4.f * M_PI * Intensity;
}
```

Finally, since point lights represent singularities that only cast incident light along a single direction, `PointLight::IsDeltaLight()` returns `true`.

⟨*PointLight Public Methods*⟩ ≡ 610
```
bool IsDeltaLight() const { return true; }
```

DistanceSquared() 65
LightSample 710
M_PI 1002
Point 63
PointLight 610
PointLight::Intensity 611
PointLight::IsDeltaLight() 611
PointLight::lightPos 611
Scene 22
Spectrum 263
Vector 57
Vector::Normalize() 63
VisibilityTester 608
VisibilityTester::SetSegment() 609

12.2.1 SPOTLIGHTS

Spotlights are a handy variation on point lights; rather than shining illumination in all directions, they emit light in a cone of directions from their position. For simplicity, we will define the spotlight in the light coordinate system to always be at position (0, 0, 0) and pointing down the $+z$ axis. To place or orient it elsewhere in the scene, the

Figure 12.3: Scene Rendered with a Spotlight. The spotlight cone smoothly cuts off illumination past a user-specified angle from the light's central axis.

`Light::WorldToLight` transformation should be set accordingly. Figure 12.3 shows a rendering of the same scene as Figure 12.2, only illuminated with a spotlight instead of a point light. The `SpotLight` class is defined in `lights/spot.h` and `lights/spot.cpp`.

⟨*SpotLight Declarations*⟩ ≡
```
class SpotLight : public Light {
public:
    ⟨SpotLight Public Methods⟩
private:
    ⟨SpotLight Private Data 613⟩
};
```

Two angles are passed to the constructor to set the extent of the `SpotLight`'s cone: the overall angular width of the cone and the angle at which falloff starts (Figure 12.4). The constructor precomputes and stores the cosines of these angles for use in the `SpotLight`'s methods.

⟨*SpotLight Method Definitions*⟩ ≡
```
SpotLight::SpotLight(const Transform &light2world,
                     const Spectrum &intensity, float width, float fall)
    : Light(light2world) {
    lightPos = LightToWorld(Point(0,0,0));
    Intensity = intensity;
    cosTotalWidth = cosf(Radians(width));
    cosFalloffStart = cosf(Radians(fall));
}
```

Light 606
Light::LightToWorld 606
Light::WorldToLight 606
Point 63
Radians() 1001
Spectrum 263
SpotLight 612
SpotLight::
cosFalloffStart 613
SpotLight::cosTotalWidth 613
SpotLight::Intensity 613
SpotLight::lightPos 613
Transform 76

⟨*SpotLight Private Data*⟩ ≡ 612
```
Point lightPos;
Spectrum Intensity;
float cosTotalWidth, cosFalloffStart;
```

The `SpotLight::Sample_L()` method is almost identical to `PointLight::Sample_L()`, except that it also calls the `Falloff()` method, which computes the distribution of light accounting for the spotlight cone. This computation is encapsulated in a separate method since other `SpotLight` methods will need to perform it as well.

⟨*SpotLight Method Definitions*⟩ +≡
```
Spectrum SpotLight::Sample_L(const Point &p, float pEpsilon,
        const LightSample &ls, float time, Vector *wi,
        float *pdf, VisibilityTester *visibility) const {
    *wi = Normalize(lightPos - p);
    *pdf = 1.f;
    visibility->SetSegment(p, pEpsilon, lightPos, 0., time);
    return Intensity * Falloff(-*wi) / DistanceSquared(lightPos, p);
}
```

To compute the spotlight's strength for a receiving point p, this first step is to compute the cosine of the angle between the vector from the spotlight origin to p and the vector along the center of the spotlight's cone. To compute the cosine of the offset angle to a point p, we have (Figure 12.4)

$$\begin{aligned}\cos\theta &= \left(\mathrm{p} \mathbin{\widehat{-}} (0,0,0)\right) \cdot (0,0,1) \\ &= \mathrm{p}_z / \|\mathrm{p}\|.\end{aligned}$$

This value is then compared to the cosines of the falloff and overall width angles to see where the point lies with respect to the spotlight cone. We can trivially determine that points with a cosine greater than the cosine of the falloff angle are inside the cone

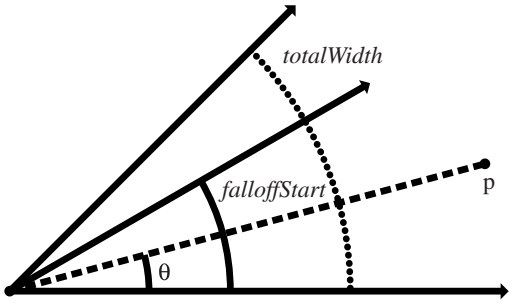


Figure 12.4: Spotlights are defined by two angles, *falloffStart* and *totalWidth*. Objects inside the inner cone of angles, up to *falloffStart*, are fully illuminated by the light. The directions between *falloffStart* and *totalWidth* are a transition zone that ramps down from full illumination to no illumination, such that points outside the *totalWidth* cone aren't illuminated at all. The cosine of the angle between the vector to a point p and the spotlight axis, θ, can easily be computed with a dot product.

`DistanceSquared()` 65
`LightSample` 710
`Point` 63
`PointLight::Sample_L()` 611
`Spectrum` 263
`SpotLight` 612
`SpotLight::Falloff()` 614
`SpotLight::Intensity` 613
`SpotLight::lightPos` 613
`SpotLight::Sample_L()` 713
`Vector` 57
`Vector::Normalize()` 63
`VisibilityTester` 608
`VisibilityTester::SetSegment()` 609

receiving full illumination, and points with cosine less than the width angle's cosine are completely outside the cone. (Note that the computation is slightly tricky since for $\theta \in [0, 2\pi]$, if $\theta > \theta'$, then $\cos\theta < \cos\theta'$.)

⟨*SpotLight Method Definitions*⟩ +≡
```
float SpotLight::Falloff(const Vector &w) const {
    Vector wl = Normalize(WorldToLight(w));
    float costheta = wl.z;
    if (costheta < cosTotalWidth)     return 0.;
    if (costheta > cosFalloffStart)   return 1.;
    ⟨Compute falloff inside spotlight cone 614⟩
}
```

For points inside the transition range from fully illuminated to outside of the cone, the intensity is scaled to smoothly fall off from full illumination to darkness:

⟨*Compute falloff inside spotlight cone*⟩ ≡ 614
```
float delta = (costheta - cosTotalWidth) /
              (cosFalloffStart - cosTotalWidth);
return delta*delta*delta*delta;
```

The solid angle subtended by a cone with spread angle θ is $2\pi(1 - \cos\theta)$. Therefore, the integral over directions on the sphere that gives power from radiant intensity can be solved to compute the total power of a light that only emits illumination in a cone. For the spotlight, we can reasonably approximate the power of the light by computing the solid angle of directions that is covered by the cone with a spread angle cosine halfway between `width` and `fall`.

⟨*SpotLight Method Definitions*⟩ +≡
```
Spectrum SpotLight::Power(const Scene *) const {
    return Intensity * 2.f * M_PI *
           (1.f - .5f * (cosFalloffStart + cosTotalWidth));
}
```

12.2.2 TEXTURE PROJECTION LIGHTS

Another useful light source acts like a slide projector; it takes an image map and projects its image out into the scene. The `ProjectionLight` class uses a projective transformation to project points in the scene onto the light's projection plane based on the field of view angle given to the constructor (Figure 12.5). Its implementation is in `lights/projection.h` and `lights/projection.cpp`. The use of this light in the lighting example scene is shown in Figure 12.6.

⟨*ProjectionLight Declarations*⟩ ≡
```
class ProjectionLight : public Light {
public:
    ⟨ProjectionLight Public Methods⟩
private:
    ⟨ProjectionLight Private Data 616⟩
};
```

Light 606
Light::WorldToLight 606
M_PI 1002
ProjectionLight 614
Scene 22
Spectrum 263
SpotLight 612
SpotLight::cosFalloffStart 613
SpotLight::cosTotalWidth 613
SpotLight::Intensity 613
Vector 57
Vector::Normalize() 63

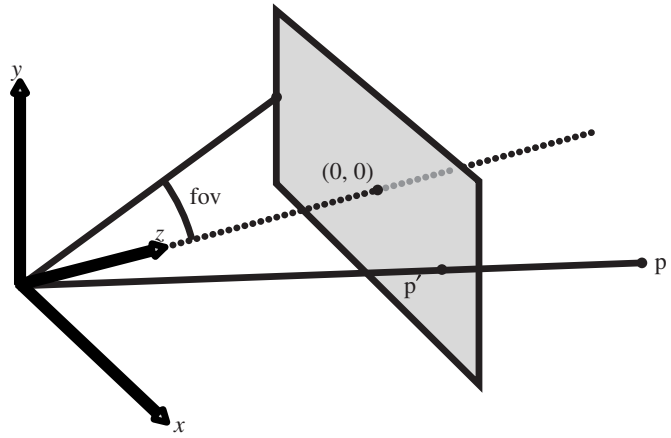


Figure 12.5: The Basic Setting for Projection Light Sources. A point p in the light's coordinate system is projected onto the plane of the image using the light's projection matrix.

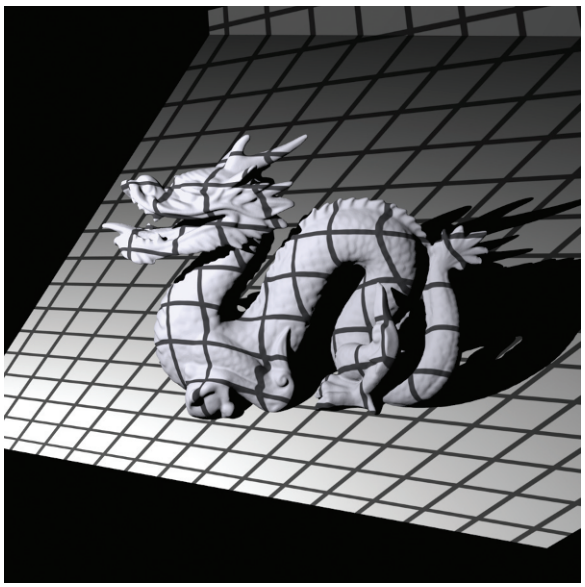


Figure 12.6: Scene Rendered with a Projection Light Using a Grid Texture Map. The projection light acts like a slide projector, projecting an image onto objects in the scene.

⟨*ProjectionLight Method Definitions*⟩ ≡
```
ProjectionLight::ProjectionLight(const Transform &light2world,
        const Spectrum &intensity, const string &texname,
        float fov)
    : Light(light2world) {
    lightPos = LightToWorld(Point(0,0,0));
    Intensity = intensity;
    ⟨Create ProjectionLight MIP-map 616⟩
    ⟨Initialize ProjectionLight projection matrix 616⟩
    ⟨Compute cosine of cone surrounding projection directions 617⟩
}
```

This light could use a `Texture` to represent the light projection distribution so that procedural projection patterns could be used. However, having a precise representation of the projection function, as is available by using an image in a `MIPMap`, is useful for being able to sample the projection distribution using Monte Carlo techniques, so we will use that representation in the implementation here. Note that the projected image is explicitly stored as a `RGBSpectrum`, even if full-spectral rendering is being performed. Unless the image map being used has full spectral data, storing full `SampledSpectrum` values in this case only wastes memory; whether an RGB color is converted to a `SampledSpectrum` before the `MIPMap` is created or after the `MIPMap` returns a value from its `Lookup()` routine gives the same result in either case.

⟨*Create* `ProjectionLight` *MIP-map*⟩ ≡ 616
```
int width, height;
RGBSpectrum *texels = ReadImage(texname, &width, &height);
if (texels)
    projectionMap = new MIPMap<RGBSpectrum>(width, height, texels);
else
    projectionMap = NULL;
delete[] texels;
```

⟨*ProjectionLight Private Data*⟩ ≡ 614
```
MIPMap<RGBSpectrum> *projectionMap;
Point lightPos;
Spectrum Intensity;
```

Similar to the `PerspectiveCamera`, the `ProjectionLight` constructor computes a projection matrix and the screen space extent of the projection.

⟨*Initialize* `ProjectionLight` *projection matrix*⟩ ≡ 616
```
float aspect = float(width) / float(height);
if (aspect > 1.f)  {
    screenX0 = -aspect; screenX1 = aspect;
    screenY0 = -1.f;    screenY1 = 1.f;
}
else {
    screenX0 = -1.f;            screenX1 = 1.f;
    screenY0 = -1.f / aspect;   screenY1 = 1.f / aspect;
}
```

Light 606
Light::LightToWorld 606
MIPMap 530
Perspective() 311
PerspectiveCamera 310
Point 63
ProjectionLight 614
ProjectionLight::hither 617
ProjectionLight::Intensity 616
ProjectionLight::lightPos 616
ProjectionLight::lightProjection 617
ProjectionLight::projectionMap 616
ProjectionLight::screenX0 617
ProjectionLight::screenX1 617
ProjectionLight::screenY0 617
ProjectionLight::screenY1 617
ProjectionLight::yon 617
ReadImage() 1004
RGBSpectrum 279
SampledSpectrum 266
Spectrum 263
Texture 519
Transform 76

```
hither = 1e-3f;
yon = 1e30f;
lightProjection = Perspective(fov, hither, yon);
```

⟨*ProjectionLight Private Data*⟩ +≡ 614
```
Transform lightProjection;
float hither, yon;
float screenX0, screenX1, screenY0, screenY1;
```

Finally, the constructor finds the cosine of the angle between the $+z$ axis and the vector to a corner of the screen window. This value is used elsewhere to define the minimal cone of directions that encompasses the set of directions in which light is projected. This cone is useful for algorithms like photon mapping that need to randomly sample rays leaving the light source. We won't derive this computation here; it is based on straightforward trigonometry.

⟨*Compute cosine of cone surrounding projection directions*⟩ ≡ 616
```
float opposite = tanf(Radians(fov) / 2.f);
float tanDiag = opposite * sqrtf(1.f + 1.f/(aspect*aspect));
cosTotalWidth = cosf(atanf(tanDiag));
```

⟨*ProjectionLight Private Data*⟩ +≡ 614
```
float cosTotalWidth;
```

Similar to the spotlight's version, `ProjectionLight::Sample_L()` calls a utility method, `ProjectionLight::Projection()`, to determine how much light is projected in the given direction. Therefore, we won't include the implementation of `Sample_L()` here.

⟨*ProjectionLight Method Definitions*⟩ +≡
```
Spectrum ProjectionLight::Projection(const Vector &w) const {
    Vector wl = WorldToLight(w);
    ⟨Discard directions behind projection light 617⟩
    ⟨Project point onto projection plane and compute light 618⟩
}
```

Because the projective transformation has the property that it projects points behind the center of projection to points in front of it, it is important to discard points with a negative z value. Therefore, the projection code immediately returns no illumination for projection points that are behind the hither plane for the projection. If this check were not done, then it wouldn't be possible to know if a projected point was originally behind the light (and therefore not illuminated) or in front of it.

⟨*Discard directions behind projection light*⟩ ≡ 617
```
if (wl.z < hither) return 0.;
```

After being projected to the projection plane, points with coordinate values outside the screen window are discarded. Points that pass this test are transformed to get (s, t) texture coordinates inside $[0, 1]^2$ for the lookup in the projection's image map.

Light::WorldToLight 606
ProjectionLight 614
ProjectionLight::cosTotalWidth 617
ProjectionLight::hither 617
ProjectionLight::Projection() 617
Radians() 1001
Spectrum 263
Transform 76
Vector 57

⟨*Project point onto projection plane and compute light*⟩ ≡ 617
```
Point Pl = lightProjection(Point(wl.x, wl.y, wl.z));
if (Pl.x < screenX0 || Pl.x > screenX1 ||
    Pl.y < screenY0 || Pl.y > screenY1) return 0.;
if (!projectionMap) return 1;
float s = (Pl.x - screenX0) / (screenX1 - screenX0);
float t = (Pl.y - screenY0) / (screenY1 - screenY0);
return Spectrum(projectionMap->Lookup(s, t), SPECTRUM_ILLUMINANT);
```

The total power of this light is approximated as a spotlight that subtends the same angle as the diagonal of the projected image, scaled by the average intensity in the image map. This approximation becomes increasingly inaccurate as the projected image's aspect ratio becomes less square, for example, and it doesn't account for the fact that texels toward the edges of the image map subtend a larger solid angle than texels in the middle when projected with a perspective projection. Nevertheless, it's a reasonable first-order approximation. Note it is explicitly specified that the `RGBSpectrum` value passed to the `Spectrum` constructor represents an illuminant's SPD and not that of a reflectance. (Recall from Section 5.2.2 that different matching functions are used for converting from RGB to SPDs for illuminants versus reflectances.)

⟨*ProjectionLight Method Definitions*⟩ +≡
```
Spectrum ProjectionLight::Power(const Scene *) const {
    return Spectrum(projectionMap->Lookup(.5f, .5f, .5f),
                    SPECTRUM_ILLUMINANT) *
        Intensity * 2.f * M_PI * (1.f - cosTotalWidth);
}
```

12.2.3 GONIOPHOTOMETRIC DIAGRAM LIGHTS

A *goniophotometric diagram* describes the angular distribution of luminance from a point light source; it is widely used in illumination engineering to characterize lights. Figure 12.7 shows an example of a goniophotometric diagram in two dimensions. In this section, we'll implement a light source that uses goniophotometric diagrams encoded in 2D image maps to describe the emission distribution of the light. The implementation is very similar to the point light sources defined previously in this section; it scales the intensity based on the outgoing direction according to the goniophotometric diagram's values. Figure 12.8 shows a few goniophotometric diagrams encoded as image maps, and Figure 12.9 shows a scene rendered with a light source that uses one of these images to modulate its directional distribution of illumination.

⟨*GonioPhotometricLight Declarations*⟩ ≡
```
class GonioPhotometricLight : public Light {
public:
    ⟨GonioPhotometricLight Public Methods 621⟩
private:
    ⟨GonioPhotometricLight Private Data 620⟩
};
```

Light 606
MIPMap::Lookup() 540
M_PI 1002
Point 63
ProjectionLight::cosTotalWidth 617
ProjectionLight::Intensity 616
ProjectionLight::lightProjection 617
ProjectionLight::projectionMap 616
ProjectionLight::screenX0 617
ProjectionLight::screenX1 617
ProjectionLight::screenY0 617
ProjectionLight::screenY1 617
RGBSpectrum 279
Scene 22
Spectrum 263
SPECTRUM_ILLUMINANT 277

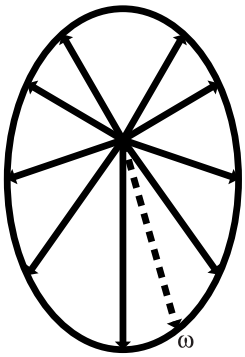


Figure 12.7: An Example of a Goniophotometric Diagram Specifying an Outgoing Light Distribution from a Point Light Source in 2D. The emitted intensity is defined in a fixed set of directions on the unit sphere, and the intensity for a given outgoing direction ω is found by interpolating the intensities of the adjacent samples.

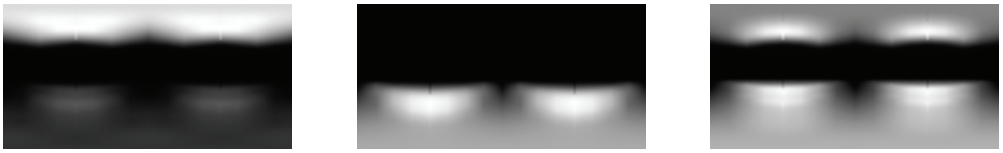


Figure 12.8: Goniophotometric Diagrams for Real-World Light Sources, Encoded as Image Maps with a Parameterization Based on Spherical Coordinates. (a) A light that mostly illuminates in its up direction, with only a small amount of illumination in the down direction. (b) A light that mostly illuminates in the down direction. (c) A light that casts illumination both above and below.

The `GonioPhotometricLight` constructor takes a base intensity and an image map that scales the intensity based on the angular distribution of light.

GonioPhotometricLight 618
GonioPhotometricLight::
Intensity 620
GonioPhotometricLight::
lightPos 620
Light 606
Light::LightToWorld 606
MIPMap 530
Point 63
ProjectionLight 614
RGBSpectrum 279
Spectrum 263
Transform 76

⟨*GonioPhotometricLight Method Definitions*⟩ ≡

```
GonioPhotometricLight::GonioPhotometricLight(const Transform &light2world,
        const Spectrum &intensity, const string &texname)
    : Light(light2world) {
    lightPos = LightToWorld(Point(0,0,0));
    Intensity = intensity;
    ⟨Create mipmap for GonioPhotometricLight 620⟩
}
```

Like `ProjectionLight`, `GonioPhotometricLight` constructs a `MIPMap` of the distribution's image map, also always using `RGBSpectrum` values.

Figure 12.9: Scene Rendered Using a Goniophotometric Diagram from Figure 12.8. Even though a point light source is the basis of this light, including the directional variation of a realistic light improves the visual realism of the rendered image.

⟨*Create* mipmap *for* GonioPhotometricLight⟩ ≡ 619
```
int width, height;
RGBSpectrum *texels = ReadImage(texname, &width, &height);
if (texels) {
    mipmap = new MIPMap<RGBSpectrum>(width, height, texels);
    delete[] texels;
}
else mipmap = NULL;
```

⟨*GonioPhotometricLight Private Data*⟩ ≡ 618
```
Point lightPos;
Spectrum Intensity;
MIPMap<RGBSpectrum> *mipmap;
```

The GonioPhotometricLight::Sample_L() method is not shown here. It is essentially identical to the SpotLight::Sample_L() and ProjectionLight::Sample_L() methods that use a helper function to scale the amount of radiance. It assumes that the scale texture is encoded using spherical coordinates, so that the given direction needs to be converted to θ and ϕ values and scaled to $[0, 1]$ before being used to index into the texture. Goniophotometric diagrams are usually defined in a coordinate space where the y axis is up, whereas the spherical coordinate utility routines assume that z is up, so y and z are swapped before doing the conversion.

GonioPhotometricLight::
mipmap 620
MIPMap 530
Point 63
ProjectionLight::
Sample_L() 617
ReadImage() 1004
RGBSpectrum 279
Spectrum 263
SpotLight::Sample_L() 713

⟨*GonioPhotometricLight Public Methods*⟩ ≡ 618
```
Spectrum Scale(const Vector &w) const {
    Vector wp = Normalize(WorldToLight(w));
    swap(wp.y, wp.z);
    float theta = SphericalTheta(wp);
    float phi   = SphericalPhi(wp);
    float s = phi * INV_TWOPI, t = theta * INV_PI;
    return (mipmap == NULL) ? 1.f :
           Spectrum(mipmap->Lookup(s, t, SPECTRUM_ILLUMINANT));
}
```

The `Power()` method's computation is inaccurate because the spherical coordinate parameterization of directions has various distortions, particularly near the $+z$ and $-z$ directions. Again, this error is acceptable for the uses of this method in `pbrt`.

⟨*GonioPhotometricLight Method Definitions*⟩ +≡
```
Spectrum GonioPhotometricLight::Power(const Scene *) const {
    return 4.f * M_PI * Intensity *
        Spectrum(mipmap->Lookup(.5f, .5f, .5f), SPECTRUM_ILLUMINANT);

}
```

12.3 DISTANT LIGHTS

Another useful light source type is the *distant light*, also known as a *directional light*. It describes an emitter that deposits illumination from the same direction at every point in space. Such a light is also called a point light "at infinity," since, as a point light becomes progressively farther away, it acts more and more like a directional light. For example, the sun (as considered from Earth) can be thought of as a directional light source. Although it is actually an area light source, the illumination effectively arrives at Earth in parallel beams because it is so far away. The `DistantLight`, implemented in the files `lights/distant.h` and `lights/distant.cpp`, implements a directional source.

DistantLight 621
DistantLight::L 622
DistantLight::lightDir 622
GonioPhotometricLight::Intensity 620
GonioPhotometricLight::mipmap 620
INV_PI 1002
INV_TWOPI 1002
Light 606
Light::LightToWorld 606
Light::WorldToLight 606
MIPMap::Lookup() 540
M_PI 1002
Scene 22
Spectrum 263
SPECTRUM_ILLUMINANT 277
SphericalPhi() 292
SphericalTheta() 292
Transform 76
Vector 57
Vector::Normalize() 63

⟨*DistantLight Declarations*⟩ ≡
```
class DistantLight : public Light {
public:
    ⟨DistantLight Public Methods⟩
private:
    ⟨DistantLight Private Data 622⟩
};
```

⟨*DistantLight Method Definitions*⟩ ≡
```
DistantLight::DistantLight(const Transform &light2world,
        const Spectrum &radiance, const Vector &dir)
    : Light(light2world) {
    lightDir = Normalize(LightToWorld(dir));
    L = radiance;
}
```

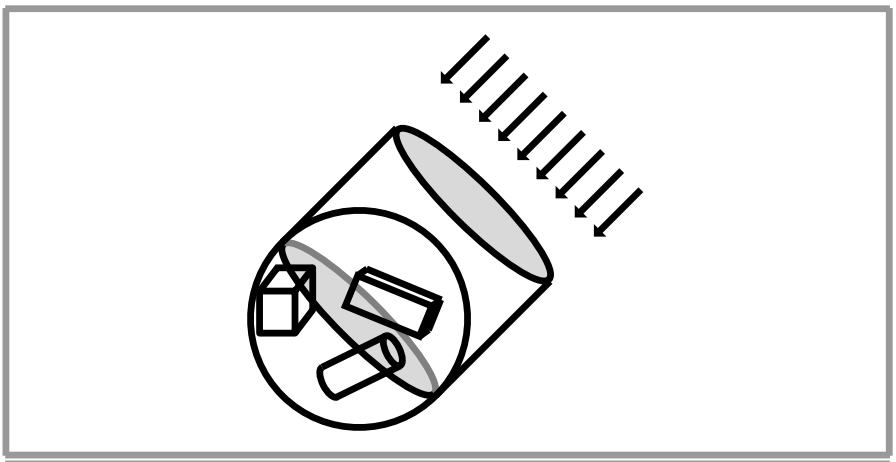


Figure 12.10: An approximation of the power emitted by a distant light into a given scene can be obtained by finding the sphere that bounds the scene, computing the area of an inscribed disk, and computing the power that arrives on the surface of that disk.

⟨*DistantLight Private Data*⟩ ≡ 621
```
Vector lightDir;
Spectrum L;
```

⟨*DistantLight Method Definitions*⟩ +≡
```
Spectrum DistantLight::Sample_L(const Point &p, float pEpsilon,
        const LightSample &ls, float time, Vector *wi, float *pdf,
        VisibilityTester *visibility) const {
    *wi = lightDir;
    *pdf = 1.f;
    visibility->SetRay(p, pEpsilon, *wi, time);
    return L;
}
```

The distant light is unusual in that the amount of power it emits is related to the spatial extent of the scene. In fact, it is proportional to the area of the scene receiving light. To see why this is so, consider a disk of area A being illuminated by a distant light with emitted radiance L where the incident light arrives along the disk's normal direction. The total power reaching the disk is $\Phi = AL$. As the size of the receiving surface varies, power varies proportionally.

To find the emitted power for a `DistantLight`, it's impractical to compute the total surface area of the objects that are visible to the light. Instead, we will approximate this area with a disk inside the scene's bounding sphere oriented in the light's direction (Figure 12.10). This will always overestimate the actual area, but is sufficient for the needs of code elsewhere in the system.

DistantLight 621
DistantLight::L 622
DistantLight::lightDir 622
LightSample 710
Point 63
Spectrum 263
Vector 57
VisibilityTester 608
VisibilityTester::
SetRay() 609

⟨*DistantLight Method Definitions*⟩ +≡
```
Spectrum DistantLight::Power(const Scene *scene) const {
    Point worldCenter;
    float worldRadius;
    scene->WorldBound().BoundingSphere(&worldCenter, &worldRadius);
    return L * M_PI * worldRadius * worldRadius;
}
```

12.4 AREA LIGHTS

Area lights are light sources defined by one or more Shapes that emit light from their surface, with some directional distribution of radiance at each point on the surface. In general, computing radiometric quantities related to area lights requires computing integrals over the surface of the light that often can't be computed in closed form. This issue is addressed with the Monte Carlo integration techniques of Chapter 14. The reward for this complexity (and computational expense) is soft shadows and more realistic lighting effects, rather than the hard shadows and stark lighting that come from point lights. (See Figure 12.11. Also compare Figure 12.12 to Figure 12.2.)

The AreaLight class is an abstract base class that inherits from Light. Implementations of area lights should inherit from it.

⟨*Light Declarations*⟩ +≡
```
class AreaLight : public Light {
public:
    ⟨AreaLight Interface 625⟩
};
```

Figure 12.11: Wider View of the Lighting Example Scene. The dragon is illuminated by a disk area light source directly above it.

AreaLight 623
BBox::BoundingSphere() 74
DistantLight::L 622
Light 606
M_PI 1002
Point 63
Scene 22
Scene::WorldBound() 24
Shape 108
Spectrum 263

(a)

(b)

Figure 12.12: Dragon Model Illuminated by Disk Area Lights. (a) The disk's radius is relatively small; the shadow has soft penumbrae, but otherwise the image looks similar to the one with a point light. (b) The effect of using a much larger disk: not only have the penumbrae become much larger, to the point of nearly eliminating the fully in-shadow areas, but notice how areas like the neck of the dragon and its jaw have noticeably different appearances when illuminated from a wider range of directions.

AreaLight adds a single new method to the Light interface, AreaLight::L(). Implementations are given a point on the surface of the light and the surface normal and should evaluate the area light's emitted radiance, L, in the given outgoing direction.

⟨*AreaLight Interface*⟩ ≡ 623
```
virtual Spectrum L(const Point &p, const Normal &n,
                   const Vector &w) const = 0;
```

For convenience, there is a method in the Intersection class that makes it easy to compute the emitted radiance at a surface point intersected by a ray. It gets the AreaLight pointer from the primitive and calls its L() method.

⟨*Intersection Method Definitions*⟩ +≡
```
Spectrum Intersection::Le(const Vector &w) const {
    const AreaLight *area = primitive->GetAreaLight();
    return area ? area->L(dg.p, dg.nn, w) : Spectrum(0.);
}
```

DiffuseAreaLight implements a basic area light source with a uniform spatial and directional radiance distribution. It only emits light on the side of the surface with outward-facing surface normal; there is no emission from the other side. (The Shape::ReverseOrientation value can be set to true to cause the light to be emitted from the other side of the surface instead.) DiffuseAreaLight is defined in the files lights/diffuse.h and lights/diffuse.cpp.

⟨*DiffuseAreaLight Declarations*⟩ ≡
```
class DiffuseAreaLight : public AreaLight {
public:
    ⟨DiffuseAreaLight Public Methods 626⟩
protected:
    ⟨DiffuseAreaLight Protected Data 626⟩
};
```

AreaLight 623
AreaLight::L() 625
DifferentialGeometry::p 102
DiffuseAreaLight 625
Intersection 186
Intersection::dg 186
Light 606
Normal 65
Point 63
Primitive::
  GetAreaLight() 187
Shape 108
Shape::CanIntersect() 110
Shape::
  ReverseOrientation 108
ShapeSet 626
Spectrum 263
Vector 57

The constructor caches the surface area of the light source since this area calculation may be computationally expensive. However, not all shapes can compute their surface area (one example is subdivision surfaces). More generally, some shapes can't be used as area lights directly since they don't implement all of the Shape methods that the AreaLight needs to call. For example, in the Monte Carlo sampling code in Chapter 14, DiffuseAreaLight methods implemented there will need to call Shape methods that sample points on the shape's surface; these methods cannot be implemented for some types of shapes.

Such shapes must be refined into simpler shapes until their CanIntersect() methods return true.[2] This refinement is done in the DiffuseAreaLight constructor if needed. The eventual shape or shapes that are created are stored in a container class called ShapeSet. This class implements the Shape interface, so that the DiffuseAreaLight can be written to store only a single Shape, thus simplifying its implementation.

2 We are thus overloading the semantics of the Shape::CanIntersect() method to include both the ability to compute intersections, the ability to compute surface area, and the ability to generate samples.

⟨*DiffuseAreaLight Method Definitions*⟩ ≡
```
DiffuseAreaLight::DiffuseAreaLight(const Transform &light2world,
        const Spectrum &le, int ns, const Reference<Shape> &s)
    : AreaLight(light2world, ns) {
    Lemit = le;
    shapeSet = new ShapeSet(s);
    area = shapeSet->Area();
}
```

⟨*DiffuseAreaLight Protected Data*⟩ ≡ 625
```
Spectrum Lemit;
ShapeSet *shapeSet;
float area;
```

Because this area light implementation emits light from only one side of the shape's surface, its `L()` method just makes sure that the outgoing direction lies in the same hemisphere as the normal.

⟨*DiffuseAreaLight Public Methods*⟩ ≡ 625
```
Spectrum L(const Point &p, const Normal &n, const Vector &w) const {
    return Dot(n, w) > 0.f ? Lemit : 0.f;
}
```

The `ShapeSet` class is a simple wrapper for a vector of shapes. Most of its implementation is in Section 14.6.4, where its methods related to random sampling collections of shapes for Monte Carlo integration are implemented.

⟨*ShapeSet Declarations*⟩ ≡
```
class ShapeSet {
public:
    ⟨ShapeSet Public Methods⟩
private:
    ⟨ShapeSet Private Data 626⟩
};
```

⟨*ShapeSet Private Data*⟩ ≡ 626
```
vector<Reference<Shape> > shapes;
```

The `DiffuseAreaLight::Sample_L()` method isn't as straightforward as it has been for the other light sources described so far. Specifically, at each point in the scene, radiance from area lights can be incident from many directions, not just a single direction as was the case for the other lights (Figure 12.13). This leads to the question of which direction should be chosen for this method. We will defer answering this question and providing an implementation of this method until Section 14.6, after Monte Carlo integration has been introduced.

Emitted power from an area light with uniform emitted radiance over the surface can be directly computed in closed form:

AreaLight 623
DiffuseAreaLight 625
DiffuseAreaLight::area 626
DiffuseAreaLight::Lemit 626
DiffuseAreaLight::Sample_L() 718
Dot() 60
Normal 65
Point 63
Reference 1011
Shape 108
ShapeSet 626
Spectrum 263
Transform 76
Vector 57

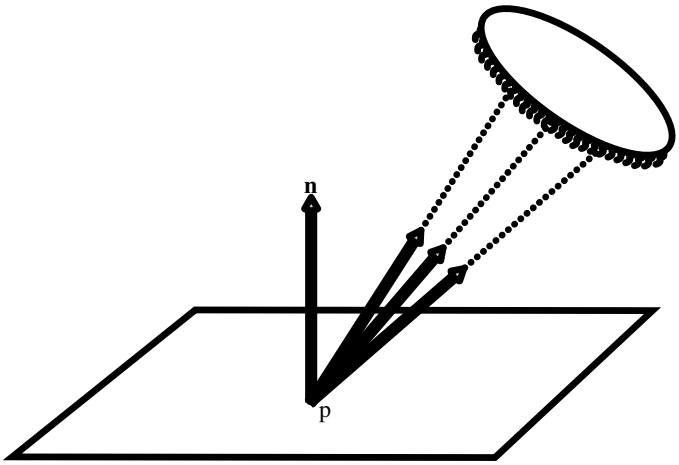


Figure 12.13: An area light casts incident illumination along many directions, rather than from a single direction.

⟨*DiffuseAreaLight Method Definitions*⟩ +≡
```
Spectrum DiffuseAreaLight::Power(const Scene *) const {
    return Lemit * area * M_PI;
}
```

Finally, area lights clearly do not represent singularities, so they return `false` from `IsDeltaLight()`:

⟨*DiffuseAreaLight Public Methods*⟩ +≡ 625
```
bool IsDeltaLight() const { return false; }
```

12.5 INFINITE AREA LIGHTS

Another useful kind of light is the infinite area light—an infinitely faraway area light source that surrounds the entire scene. One way to visualize this light is as an enormous sphere that casts light into the scene from every direction. One important use of infinite area lights is for *environment lighting*, where an image that represents illumination in an environment is used to illuminate synthetic objects as if they were in that environment. Figures 12.14 and 12.15 compare illuminating a car model with a standard area light to illuminating it with environment maps that simulate illumination from the sky at a few different times of day (the illumination maps used are shown in Figure 12.16). The increase in realism is striking. The `InfiniteAreaLight` class is implemented in `lights/infinite.h` and `lights/infinite.cpp`.

A widely used representation for light for this application is the latitude–longitude radiance map. The `EnvironmentCamera` can be used to create image maps for the light, or see the "Further Reading" section for information about techniques for capturing this

DiffuseAreaLight::area 626
DiffuseAreaLight::Lemit 626
EnvironmentCamera 318
InfiniteAreaLight 629
M_PI 1002
Scene 22
Spectrum 263

(a)

(b)

Figure 12.14: Car model (a) illuminated with an area light and a directional light and (b) illuminated with morning skylight from an environment map. Using a realistic distribution of illumination gives an image that is much more visually compelling. In particular, with illumination arriving from all directions, the glossy reflective properties of the paint are much more visually apparent; in (a) there isn't anything to reflect, so the paint inaccurately appears to be dull matte.

lighting data from real-world environments. Like the other lights, the `InfiniteAreaLight` takes a transformation matrix; here, its use is to orient the image map. It then uses spherical coordinates to map from directions on the sphere to (θ, ϕ) directions, and from there to (u, v) texture coordinates. The provided transformation thus determines which direction is "up."

`InfiniteAreaLight` 629

(a)

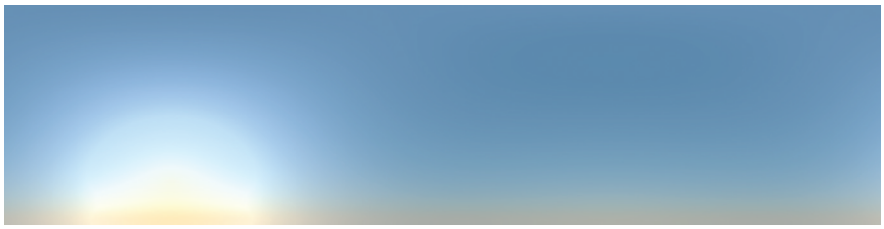
(b)

Figure 12.15: Changing just the environment map used for illumination gives quite different results in the final image: (a) using a midday skylight distribution and (b) using a sunset environment map.

⟨*InfiniteAreaLight Declarations*⟩ ≡

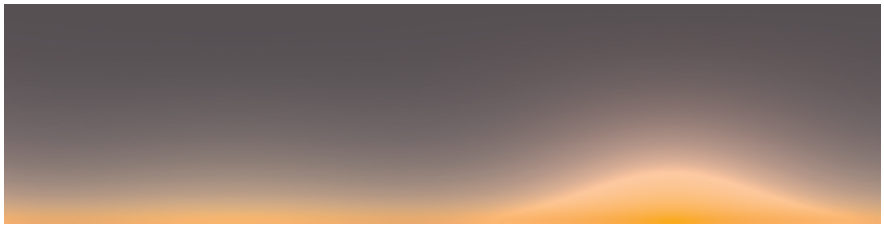
```
class InfiniteAreaLight : public Light {
public:
    ⟨InfiniteAreaLight Public Methods 631⟩
private:
    ⟨InfiniteAreaLight Private Data 631⟩
};
```

Light 606



(a)

(b)



(c)

Figure 12.16: Environment Maps Used for Illumination in Figures 12.14 and 12.15. (a) Morning, (b) midday, and (c) sunset sky. (The bottom halves of these maps aren't shown here, since they are just black pixels.)

The constructor loads the image data from disk and creates a `MIPMap` to store it. The fragment that loads the data, ⟨*Read texel data from* `texmap` *into* `texels`⟩, is straightforward and won't be included here. The other code fragment in the constructor, ⟨*Initialize sampling PDFs for infinite area light*⟩, is related to Monte Carlo sampling of `InfiniteAreaLights` and will be defined later, in Section 14.6.5.

⟨*InfiniteAreaLight Method Definitions*⟩ ≡

```
InfiniteAreaLight::InfiniteAreaLight(const Transform &light2world,
        const Spectrum &L, int ns, const string &texmap)
    : Light(light2world, ns) {
    int width = 0, height = 0;
    RGBSpectrum *texels = NULL;
    ⟨Read texel data from texmap into texels⟩
```

InfiniteAreaLight 629
InfiniteAreaLight::radianceMap 631
Light 606
MIPMap 530
RGBSpectrum 279
Spectrum 263
Transform 76

```
    radianceMap = new MIPMap<RGBSpectrum>(width, height, texels);
    delete[] texels;
    ⟨Initialize sampling PDFs for infinite area light 725⟩
}
```

⟨*InfiniteAreaLight Private Data*⟩ ≡ 629
```
MIPMap<RGBSpectrum> *radianceMap;
```

Because `InfiniteAreaLights` cast light from all directions, it's also necessary to use Monte Carlo integration to compute reflected light due to illumination from them. Therefore, the `InfiniteAreaLight::Sample_L()` method will be defined in Section 14.6.

Like directional lights, the total power from the infinite area light is related to the surface area of the scene:

⟨*InfiniteAreaLight Method Definitions*⟩ +≡
```
Spectrum InfiniteAreaLight::Power(const Scene *scene) const {
    Point worldCenter;
    float worldRadius;
    scene->WorldBound().BoundingSphere(&worldCenter, &worldRadius);
    return M_PI * worldRadius * worldRadius *
        Spectrum(radianceMap->Lookup(.5f, .5f, .5f), SPECTRUM_ILLUMINANT);
}
```

BBox::BoundingSphere() 74
InfiniteAreaLight 629
InfiniteAreaLight::radianceMap 631
InfiniteAreaLight::Sample_L() 727
INV_PI 1002
INV_TWOPI 1002
Light 606
Light::WorldToLight 606
MIPMap 530
MIPMap::Lookup() 540
M_PI 1002
Point 63
Ray::d 67
RayDifferential 69
RGBSpectrum 279
Scene 22
Scene::WorldBound() 24
Spectrum 263
SPECTRUM_ILLUMINANT 277
SphericalPhi() 292
SphericalTheta() 292
Vector 57
Vector::Normalize() 63

⟨*InfiniteAreaLight Public Methods*⟩ ≡ 629
```
bool IsDeltaLight() const { return false; }
```

Because infinite area lights need to be able to contribute radiance to rays that don't hit any geometry in the scene, we'll add a method to the base `Light` class that returns emitted radiance due to that light along a ray that didn't hit anything in the scene. (The default implementation for other lights returns no radiance.) It is the responsibility of the integrators to call this method for these rays.

⟨*Light Method Definitions*⟩ +≡
```
Spectrum Light::Le(const RayDifferential &) const {
    return Spectrum(0.);
}
```

⟨*InfiniteAreaLight Method Definitions*⟩ +≡
```
Spectrum InfiniteAreaLight::Le(const RayDifferential &r) const {
    Vector wh = Normalize(WorldToLight(r.d));
    float s = SphericalPhi(wh) * INV_TWOPI;
    float t = SphericalTheta(wh) * INV_PI;
    return Spectrum(radianceMap->Lookup(s, t), SPECTRUM_ILLUMINANT);
}
```

FURTHER READING

Warn (1983) developed early models of light sources with nonisotropic emission distributions, including the spotlight model used in this chapter. Verbeck and Greenberg (1984) also described a number of techniques for modeling light sources that are now classic parts of the light modeling toolbox. More recently, Barzel (1997) described a highly parameterized model for light sources, including many controls for controlling rate of falloff, the area of space that is illuminated, and so on. Bjorke (2001) described a number of additional controls for controlling illumination for artistic effect. (The Barzel and Bjorke approaches are not physically based, which isn't a problem in most applications outside of physically based rendering.)

Blinn and Newell (1976) first introduced the idea of environment maps and their use for simulating illumination, although they only considered illumination of specular objects. Greene (1986) further refined these ideas, considering antialiasing and different representations for environment maps. Nishita and Nakamae (1986) developed algorithms for efficiently rendering objects illuminated by hemispherical skylights and generated some of the first images that showed off that distinctive lighting effect.

Miller and Hoffman (1984) were the first to consider using arbitrary environment maps to illuminate objects with diffuse and glossy BRDFs. Debevec (1998) later extended this work and investigated issues related to capturing images of real environments. The *HDRShop* tool, available from *www.debevec.org/HDRShop*, is invaluable for converting among different environment map formats and parameterizations.

Algorithms to render soft shadows from area lights were first developed by Amanatides (1984) and Cook, Porter, and Carpenter (1984). See both Hasenfratz et al. (2003) and Akenine-Möller et al. (2008) for overviews of more recent algorithms for rendering soft shadows, particularly for interactive rendering.

The goniometric light source approximation is widely used to model area light sources in the field of illumination engineering. The rule of thumb there is that once a point is five times an area light source's radius away from it, a point light approximation has sufficient accuracy for most applications. Ashdown (1993) discusses this application in depth and proposes a more sophisticated light source representation for accurate illumination computations. A variety of file format standards have been developed for encoding goniophotometric diagrams for these applications (Illuminating Engineering Society of North America Computer Committee 1986; Stockmar 1986). Many lighting fixture manufacturers provide data in these formats on their Web sites. Another generalization of goniometric lights was suggested by Heidrich et al. (1998), who represented light sources as a 4D exitant *lightfield*—essentially a function of both position and direction.

As discussed previously, one way to reduce the time spent tracing shadow rays is to have methods like `Shape::IntersectP()` and `Primitive::IntersectP()` that just check for any occlusion along a ray without bothering to compute the geometric information at the intersection point. Other approaches for optimizing ray tracing for shadow rays include the *shadow cache*, where each light stores a pointer to the last primitive that occluded a shadow ray to the light where that primitive is checked first to see if it occludes subsequent shadow rays before the ray is passed to the acceleration structure (Haines and

Primitive::IntersectP() 186
Shape::IntersectP() 111

Greenberg 1986). Pearce (1991) pointed out that the shadow cache doesn't work well if the scene has finely tessellated geometry; it may be better to cache the voxel of BVH node that held the last occluder, for instance. (The shadow cache can similarly be defeated when multiple levels of reflection and refraction are present or when Monte Carlo ray-tracing techniques are used.) Hart, Dutré, and Greenberg (1999) developed a generalization of the shadow cache, which tracks which objects block light from particular light sources and clips their geometry against the light source geometry so that shadow rays don't need to be traced toward the parts of the light that are certain to be occluded.

A related technique, described by Haines and Greenberg (1986), is the light buffer for point light sources, where the light discretizes the directions around it and determines which objects are visible along each set of directions (and are thus potential occluding objects for shadow rays). Another effective optimization is *shaft culling*, which takes advantage of coherence among groups of rays traced in a similar set of directions (e.g., shadow rays from a single point to points on an area light source). With shaft culling, a shaft that bounds a collection of rays is computed and then the objects in the scene that penetrate the shaft are found. For all of the rays in the shaft, it is only necessary to check for intersections with those objects that intersect the shaft, and the expense of ray intersection acceleration structure traversal for each of the rays is avoided (Haines and Wallace 1994).

Woo and Amanatides (1990) classified which lights are visible, not visible, and partially visible in different parts of the scene and stored this information in a voxel-based 3D data structure, using this information to save shadow ray tests. Fernandez, Bala, and Greenberg (2002) developed a similar approach based on spatial decomposition that stores references to important blockers in each voxel and also builds up this information on demand for applications like walkthroughs.

For complex models, simplified versions of their geometry can be used for shadow ray intersections. For example, the simplification envelopes described by Cohen et al. (1996) can create a simplified mesh that bounds a given mesh from both the inside and the outside. If a ray misses the mesh that bounds a complex model from the outside, or intersects the mesh that bounds it from the inside, then no further shadow processing is necessary. Only the uncertain remaining cases need to be intersected against the full geometry. A related technique is described by Lukaszewski (2001), who uses the Minkowski sum to effectively expand primitives (or bounds of primitives) in the scene so that intersecting one ray against one of these primitives can determine if any of a collection of rays might have intersected the actual primitives.

EXERCISES

● 12.1 Shadow mapping is a technique for rendering shadows from point and distant light sources based on rendering an image from the light source's perspective that records depth in each pixel of the image and then projecting points onto the shadow map and comparing their depth to the depth of the first visible object as seen from the light in that direction. This method was first described by Williams (1978), and Reeves, Salesin, and Cook (1987) developed a number of key improvements. Modify `pbrt` to be able to render depth map images into

a file and then use them for shadow testing for lights in place of tracing shadow rays. How much faster can this be? Discuss the advantages and disadvantages of the two approaches.

❶ 12.2 Through algebraic manipulation and precomputation of one more value in the constructor, the `SpotLight::Falloff()` method can be rewritten to compute the exact same result (modulo floating-point differences) while using no square root computations and no divides (recall that the `Vector::Normalize()` method performs both a square root and a divide). Derive and implement this optimization. How much is running time improved on a spotlight-heavy scene?

❸ 12.3 The current light source implementations don't support animated transformations. Modify `pbrt` to include this functionality and render images showing off the effect of animating light positions.

❷ 12.4 Modify the `ProjectionLight` to also support orthographic projections. This variant is particularly useful even without an image map, since it gives a directional light source with a beam of user-defined extent.

❷ 12.5 Write an `AreaLight` implementation that improves on the `DiffuseAreaLight` by supporting spatially and directionally varying emitted radiance, specified via either image maps or `Textures`. Use it to render images with effects like a television illuminating a dark room or a stained-glass window lit from behind.

❷ 12.6 Read some of the papers in the "Further Reading" section that discuss the shadow cache and add this optimization to `pbrt`. Measure how much it speeds up the system for a variety of scenes. What techniques can you come up with that make it work better in the presence of multiple levels of reflection?

❸ 12.7 Modify `pbrt` to support the shaft culling algorithm (Haines and Wallace 1994). Measure the performance difference for scenes with area light sources. Make sure that your implementation still performs well even with very large light sources (like a hemispherical skylight).

AreaLight 623
DiffuseAreaLight 625
ProjectionLight 614
SpotLight::Falloff() 614
Texture 519
Vector::Normalize() 63