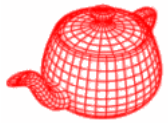
Lights

Digital Image Synthesis

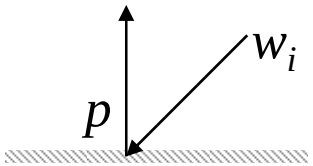
Yung-Yu Chuang

with slides by Stephen Chenney

Lights

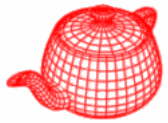


- Pbrt only supports physically-based lights, not including artistic lighting.
- **`core/light.* lights/*`**
- Essential data members:
  - **`Transform LightToWorld, WorldToLight;`**
  - **`int nSamples;`**
- Essential functions:
  - **`Spectrum Sample_L(Point &p, float pEpsilon, LightSample &ls, float time, Vector *wi, float *pdf, VisibilityTester *vis);`**
  - **`Spectrum Power(Scene *);`** approximate total power
  - **`bool IsDeltaLight();`** point/directional lights can't be sampled

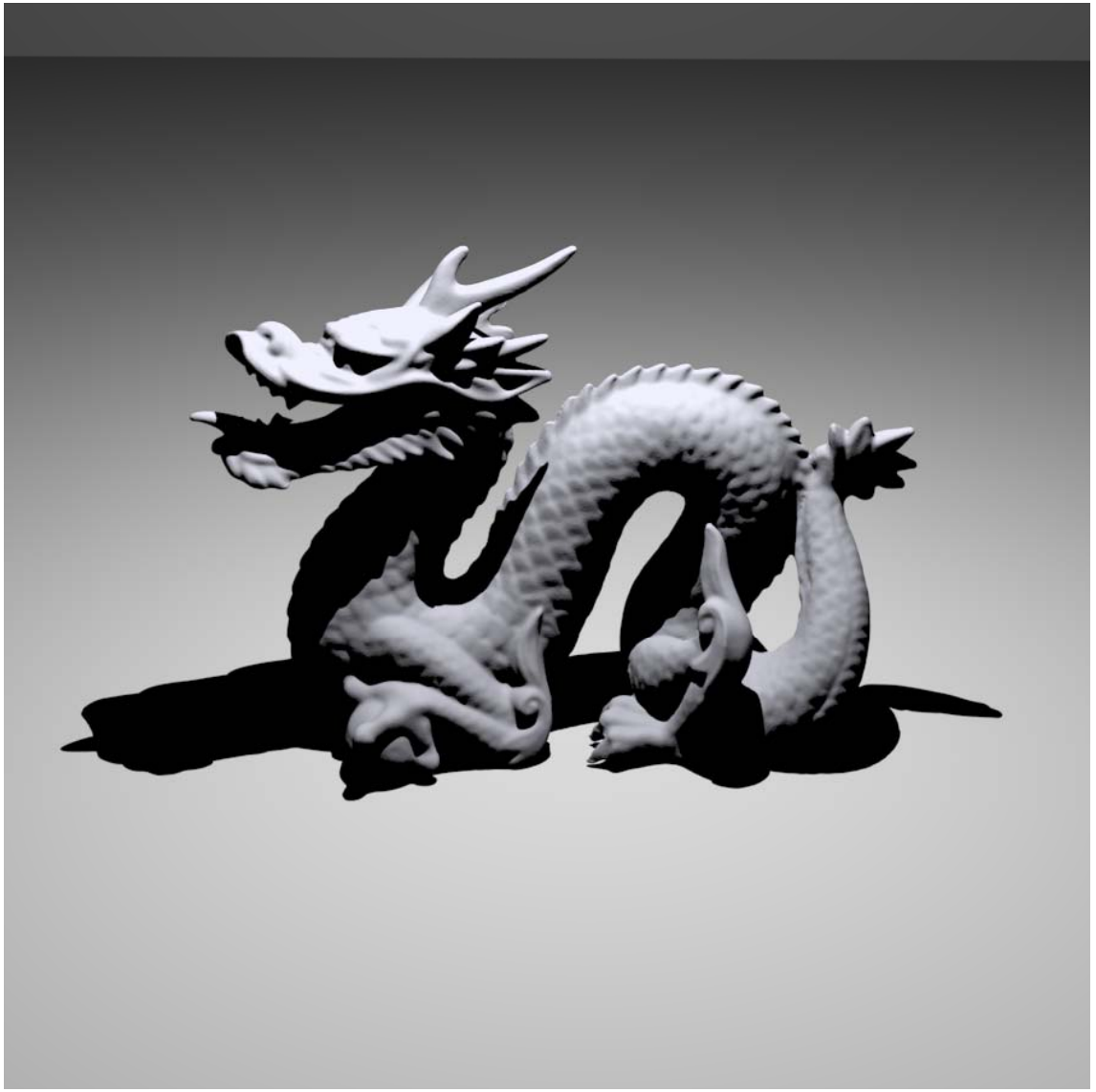


returns **`wi`** and *radiance* due to the light assuming visibility=1; initializes **`vis`**

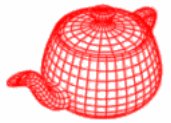
Point lights



- Isotropic
- Located at the origin



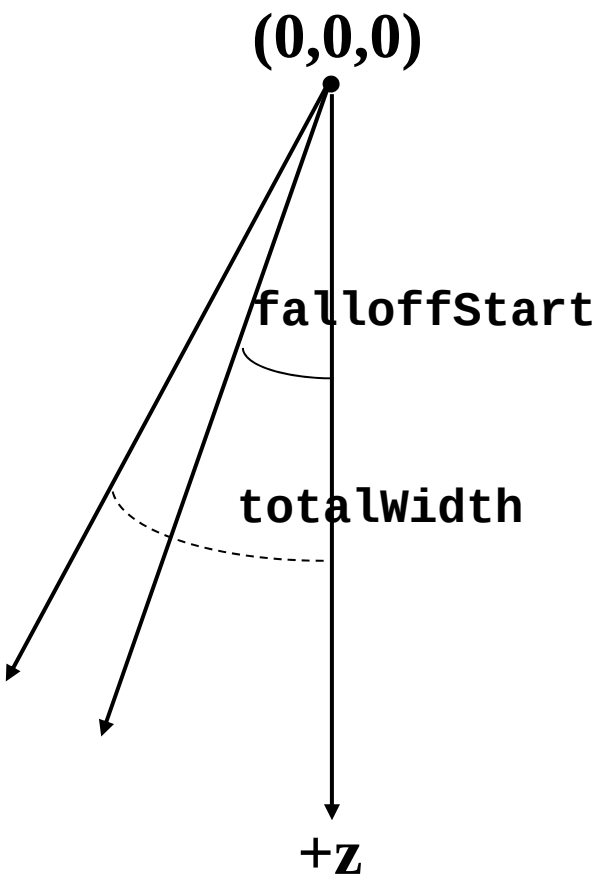
Point lights



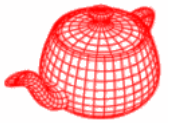
```
PointLight::PointLight(const Transform &light2world,
  const Spectrum &intensity) : Light(light2world) {
  lightPos = LightToWorld(Point(0,0,0));
  Intensity = intensity;
}
Spectrum PointLight::Sample_L(Point &p, …) {
  *wi = Normalize(lightPos - p);
  *pdf = 1.f;
  visibility->SetSegment(p,pEpsilon,lightPos,0,time);
  return Intensity / DistanceSquared(lightPos, p);
}
Spectrum Power(const Scene *) const {
  return Intensity * 4.f * M_PI;
}
```

$$I = \frac{d\Phi}{d\omega} \quad \Phi = \int_{s^2} I d\omega = 4\pi I$$

Spotlights

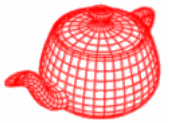


Spotlights



```
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));
}
Spectrum SpotLight::Sample_L(Point &p, …) {
  *wi = Normalize(lightPos - p);
  *pdf = 1.f;
  visibility->SetSegment(p,pEpsilon,lightPos,0,time);
  return Intensity * Falloff(-*wi)
         /DistanceSquared(lightPos,p);
}
```

Spotlights

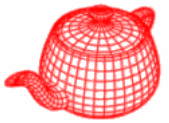


```
float SpotLight::Falloff(const Vector &w) const {
  Vector wl = Normalize(WorldToLight(w));
  float costheta = wl.z;
  if (costheta < cosTotalWidth)
    return 0.;
  if (costheta > cosFalloffStart)
    return 1.;
  float delta = (costheta - cosTotalWidth) /
                (cosFalloffStart - cosTotalWidth);
  return delta*delta*delta*delta;
}
```

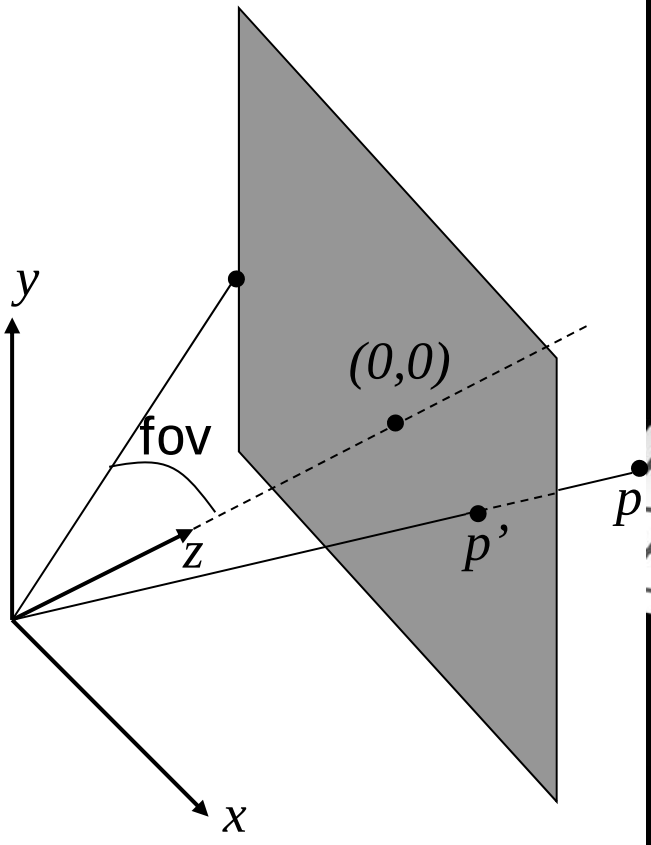
an approximation $\int_{\Omega'} d\omega = \int_{\phi=0}^{2\pi} \int_{\theta=0}^{\theta'} \sin\theta d\theta d\phi = \int_{\phi=0}^{2\pi} (1-\cos\theta') d\phi = 2\pi(1-\cos\theta')$

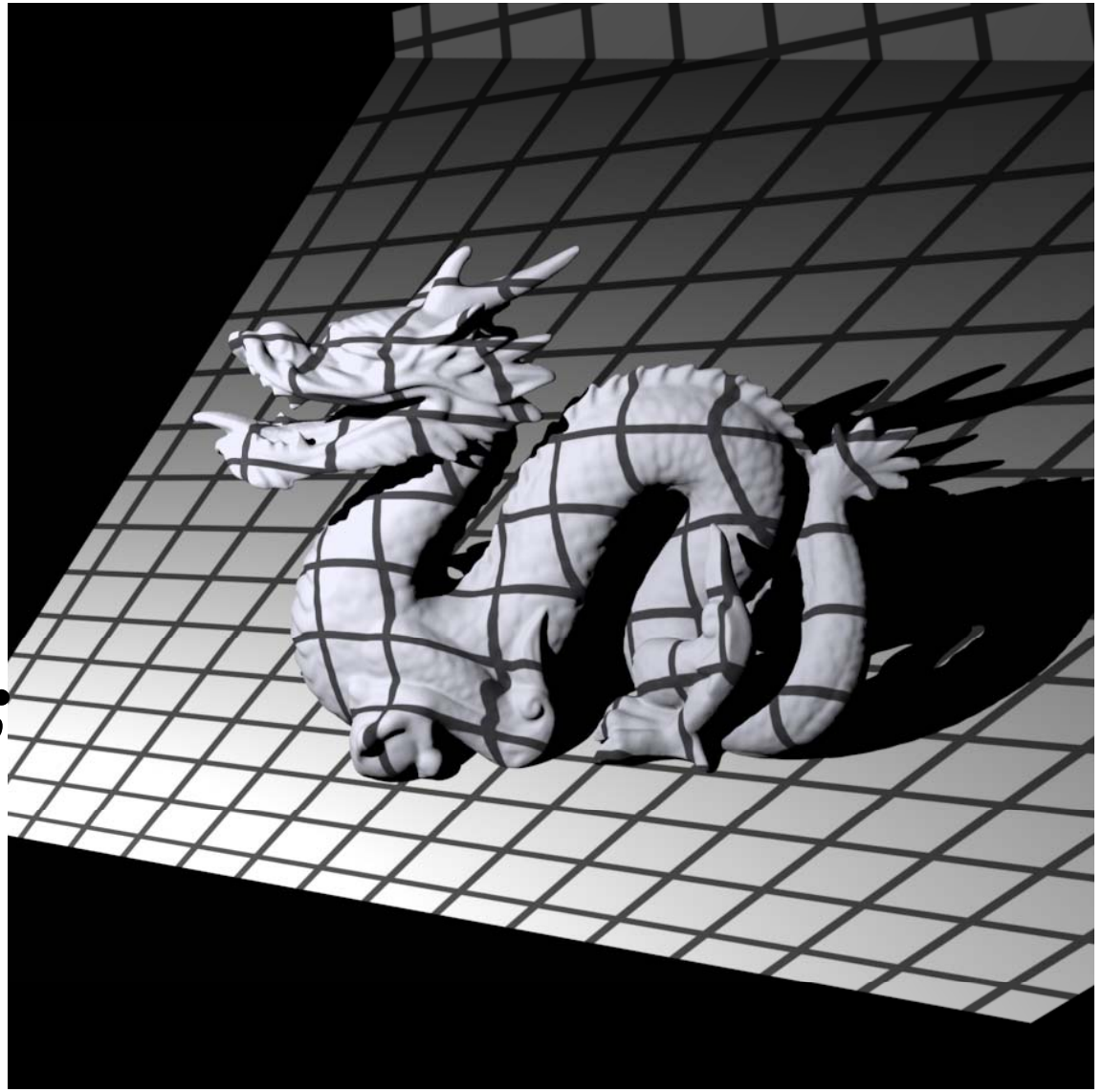
```
Spectrum Power(const Scene *) const {
  return Intensity * 2.f * M_PI *
    (1.f - .5f * (cosFalloffStart + cosTotalWidth));
}
```

Texture projection lights

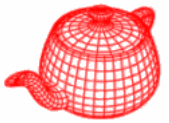


- Like a slide projector

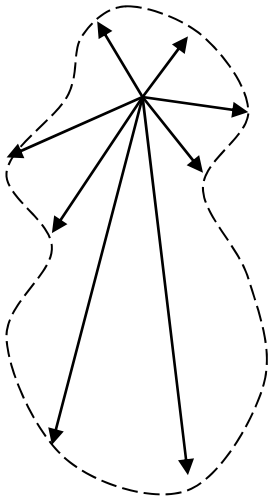


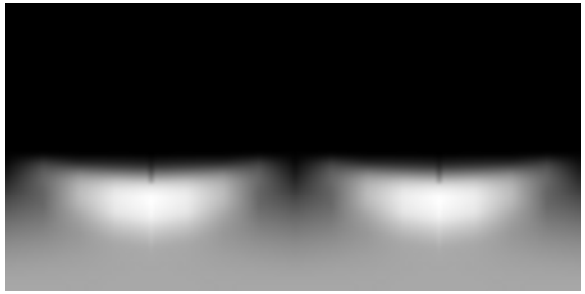


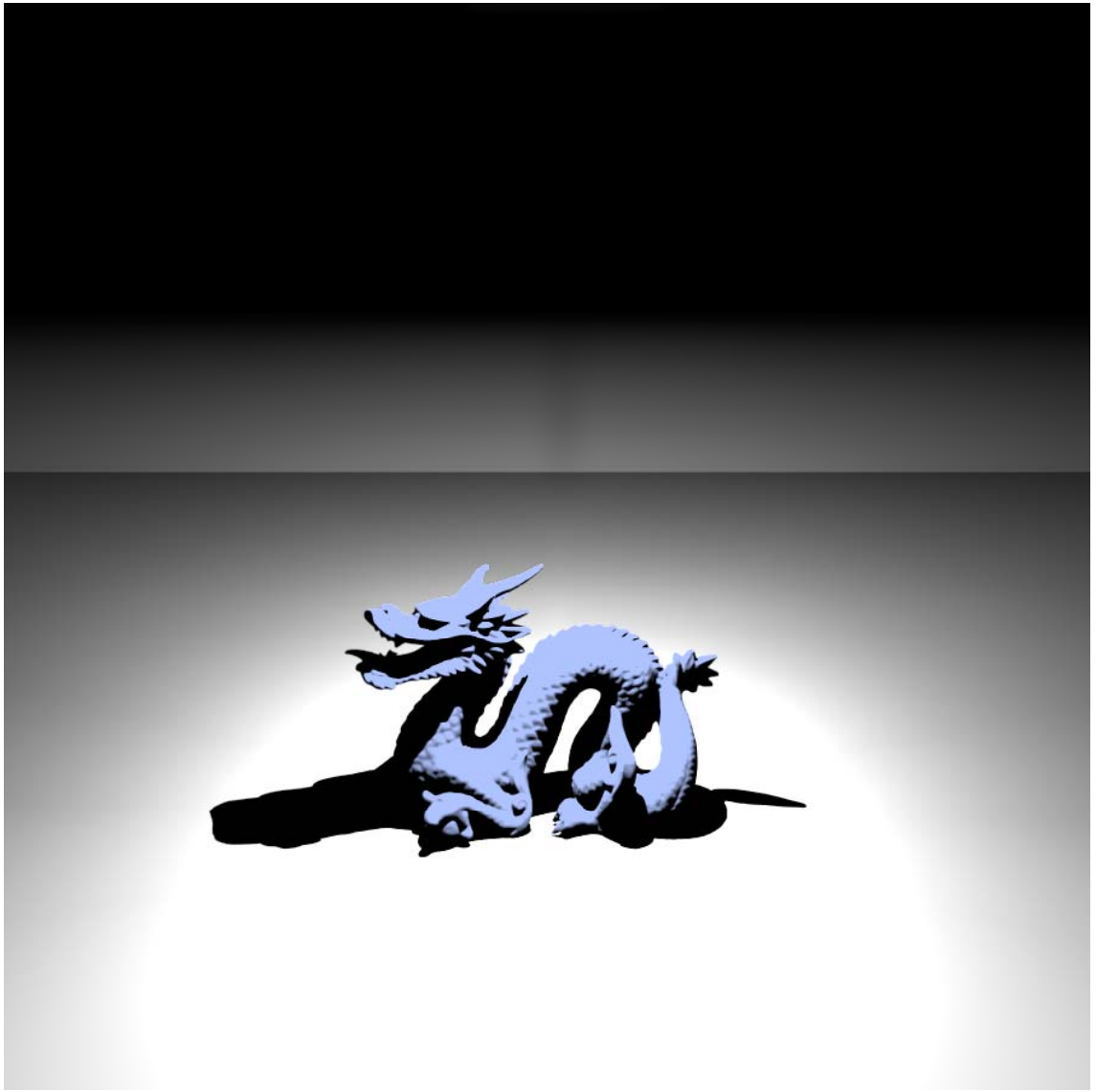
Goniophotometric light



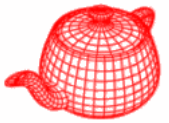
- Define a angular distribution from a point light





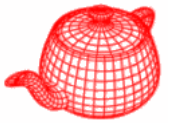


Goniophotometric light



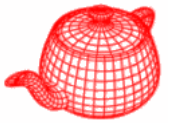
```
GonioPhotometricLight(const Transform &light2world,
  Spectrum &I, string &texname):Light(light2world) {
  lightPos = LightToWorld(Point(0,0,0));
  Intensity = I;
  int w, h;
  Spectrum *texels = ReadImage(texname, &w, &h);
  if (texels) {
    mipmap = new MIPMap<Spectrum>(w, h, texels);
    delete[] texels;
  } else mipmap = NULL;
}
Spectrum Sample_L(const Point &p, …) {
  *wi = Normalize(lightPos - p);
  *pdf = 1.f;
  visibility->SetSegment(p,pEpsilon,lightPos,0,time);
  return Intensity * Scale(-*wi)
         / DistanceSquared(lightPos, p);
}
```

Goniophotometric light



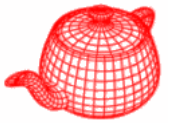
```
Spectrum Scale(const Vector &w) const {
  Vector wp = Normalize(WorldToLight(w));
  swap(wp.y, wp.z);
  float theta = SphericalTheta(wp);
  float phi   = SphericalPhi(wp);
  float s = phi * INV_TWOPI, t = theta * INV_PI;
  return (mipmap == NULL) ? 1.f :
    Spectrum(mipmap->Lookup(s,t,SPECTRUM_ILLUMINANT));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

Spectrum Power(const Scene *) const {
  return 4.f * M_PI * Intensity *
    Spectrum(mipmap ? mipmap->Lookup(.5f, .5f, .5f)
                    : 1.f, SPECTRUM_ILLUMINANT);}
```

Point lights



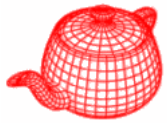
- The above four lights, point light, spotlight, texture light and goniophotometric light are essentially point lights with different energy distributions.

Directional lights

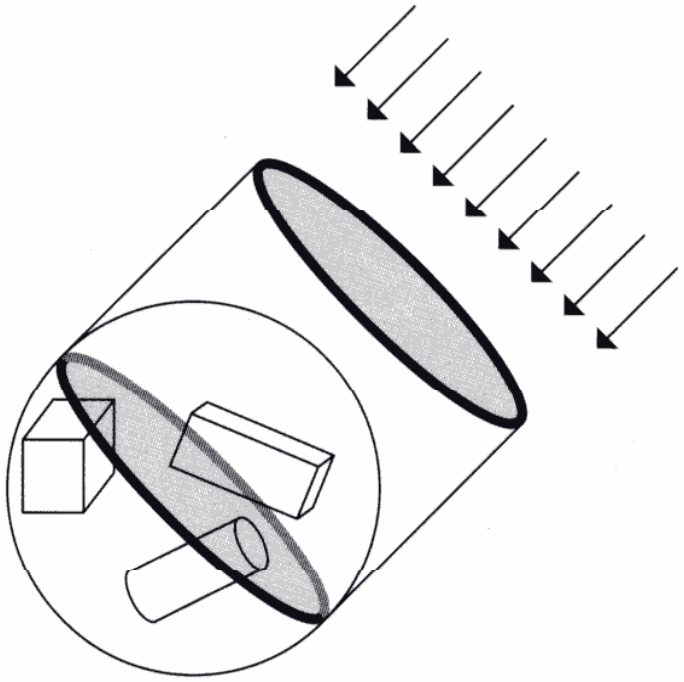


```
DistantLight::DistantLight(Transform &light2world,
Spectrum &radiance, Vector &dir):Light(light2world) {
  lightDir = Normalize(LightToWorld(dir));
  L = radiance;
}
Spectrum DistantLight::Sample_L(Point &p, …) const {
  wi = lightDir;
  *pdf = 1.f;
  visibility->SetSegment(p,pEpsilon,lightPos,0,time);
  return L;
}
```

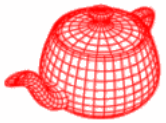
Directional lights



```
Spectrum Power(const Scene *scene) {
  Point worldCenter;
  float worldRadius;
  scene->WorldBound().BoundingSphere(&worldCenter,
  &worldRadius);
  return L * M_PI * worldRadius * worldRadius;
}
```

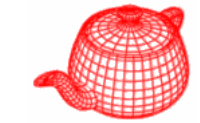


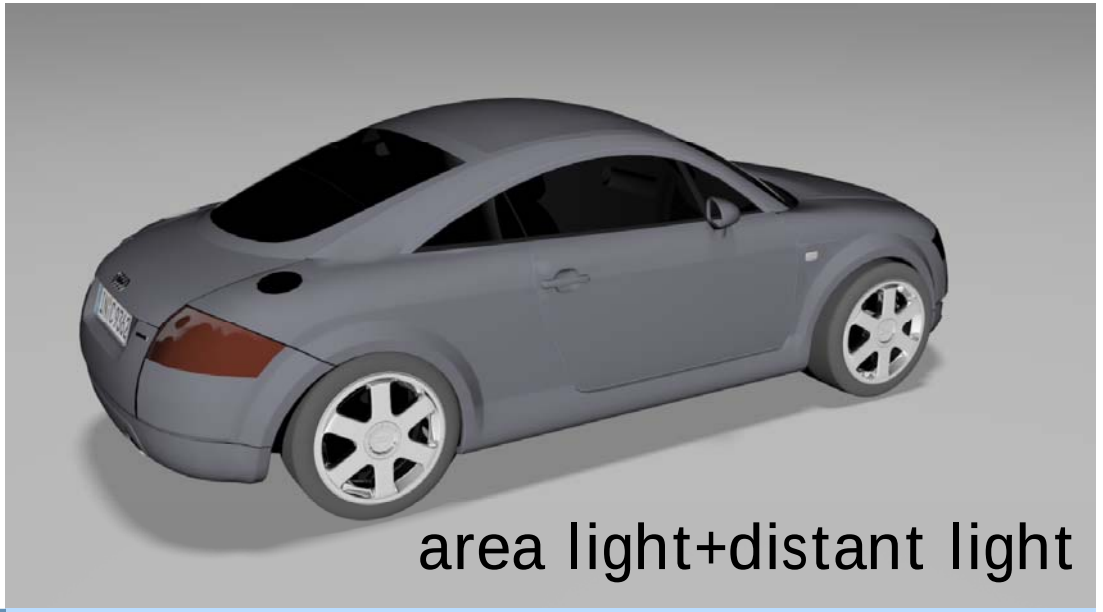
Area light

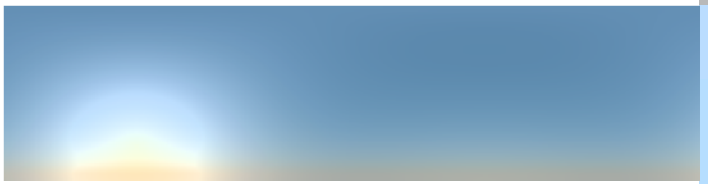


- Defined by a **shape**
- Uniform over the surface
- Single-sided
- **`Sample_L`** isn't straightforward because a point could have contributions from multiple directions (chap14).

Infinite area light (environment light)







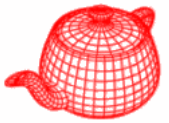
morning skylight

Infinite area light (environment light)

midday skylight

sunset skylight

Infinite area light

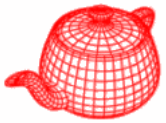


```
InfiniteAreaLight(Transform &light2world,Spectrum &L,
  int ns, string &texmap) : Light(light2world, ns) {
  <read texel data from texmap>
  <initialize sampling PDFs from infinite area light>
}
```

Sample_L and <initialize sampling PDFs …>

will be discussed after introducing Monte Carlo method

Infinite area light



```
Spectrum InfiniteAreaLight::Power(const Scene *scene)
{
  Point worldCenter;  float worldRadius;
  scene->WorldBound().BoundingSphere(&worldCenter,
                                     &worldRadius);
  return M_PI * worldRadius * worldRadius *
         Spectrum(radianceMap->Lookup(.5f,.5f,.5f),…);
}
```

for those rays which miss the scene

```
Spectrum Le(const RayDifferential &r) {
  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);
}
```