

Lights

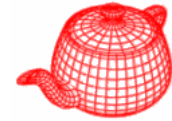
Digital Image Synthesis

Yung-Yu Chuang

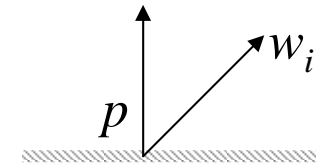
11/30/2006

with slides by Stephen Cheney

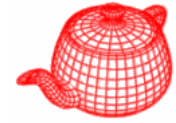
Lights



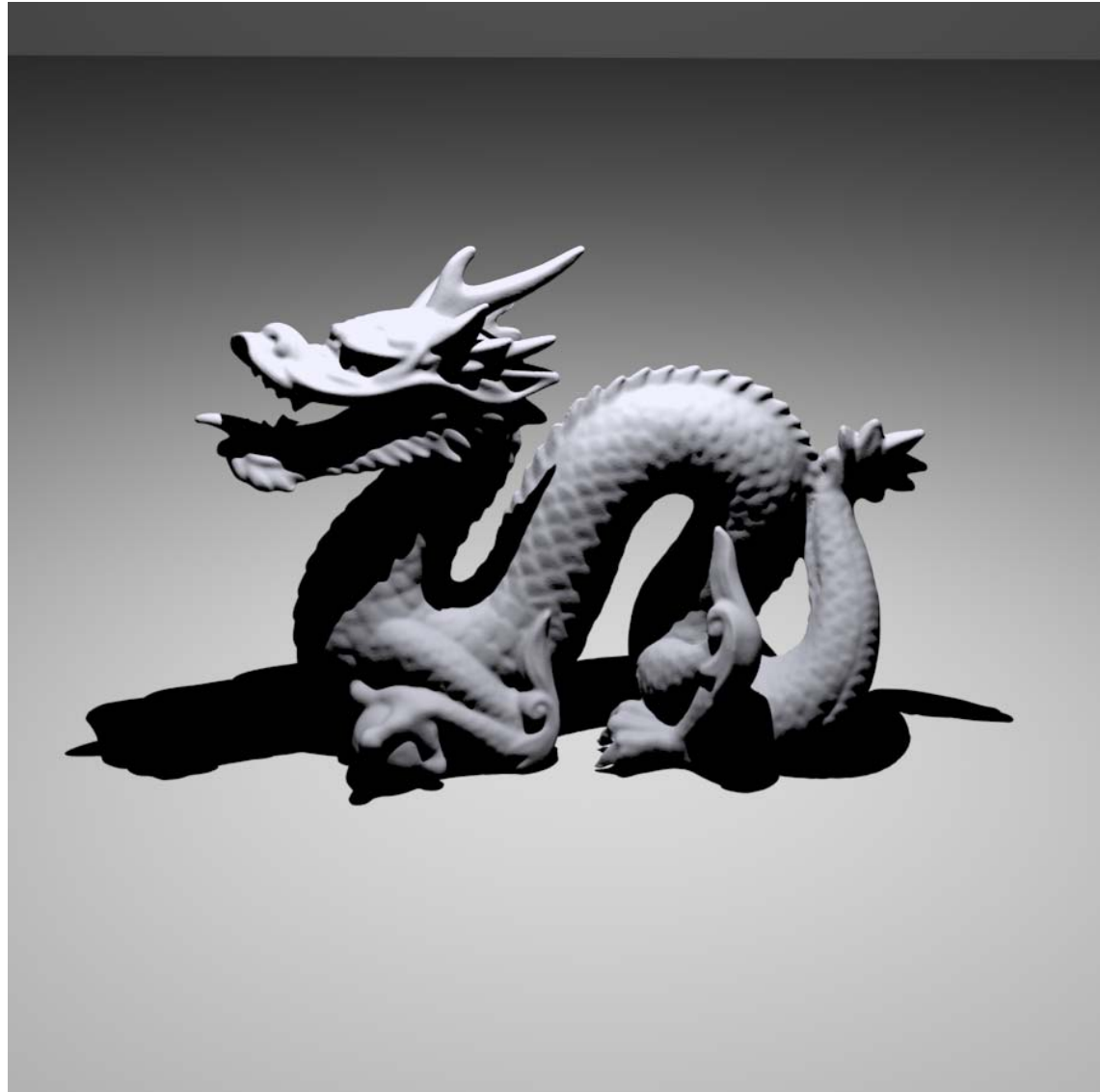
- An abstract interface for lights.
- Pbrt only supports physically-based lights, not including artistic lighting. ●
- `core/light.* lights/*`
- Essential data members:
 - `Transform LightToWorld, WorldToLight;`
 - `int nSamples;`
- Essential functions: *returns **w_i** and **radiance** due to the light assuming visibility=1; initializes **vis***
 - `Spectrum Sample_L(Point &p, Vector *wi, VisibilityTester *vis);`
 - `Spectrum Power(Scene *);` *approximate total power*
 - `bool IsDeltaLight();` *point/directional lights can't be sampled*



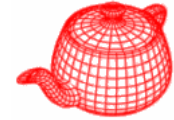
Point lights



- Isotropic
- Located at the origin



Point lights



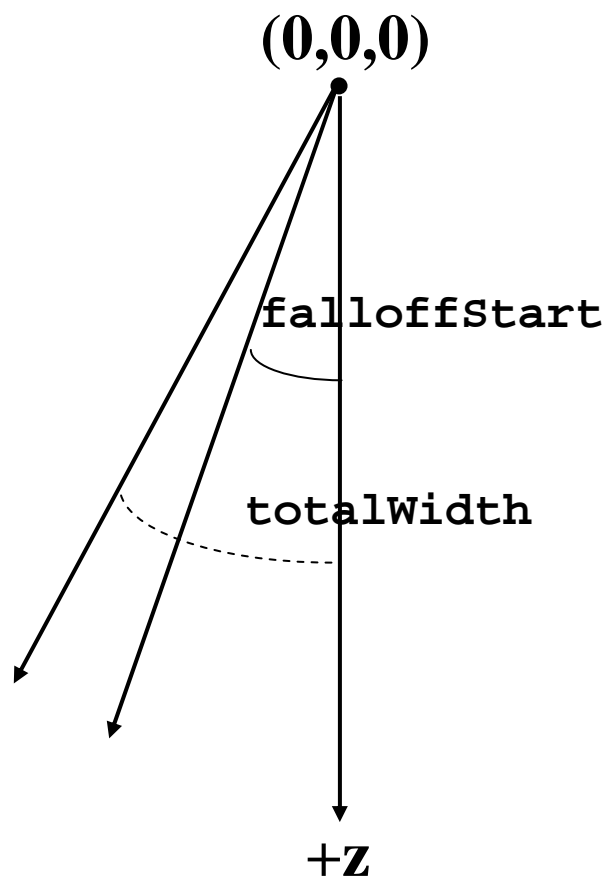
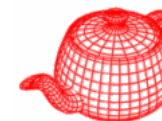
```
PointLight::PointLight(const Transform &light2world,  
    const Spectrum &intensity) : Light(light2world) {  
    lightPos = LightToWorld(Point(0,0,0));  
    Intensity = intensity;  
}
```

```
Spectrum PointLight::Sample_L(Point &p, Vector *wi,  
    VisibilityTester *visibility) {  
    *wi = Normalize(lightPos - p);  
    visibility->SetSegment(p, lightPos);  
    return Intensity / DistanceSquared(lightPos, p);  
}
```

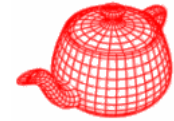
```
Spectrum Power(const Scene *) const {  
    return Intensity * 4.f * M_PI;  
}
```

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

Spotlights



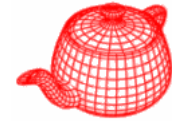
Spotlights



```
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, Vector *wi,
    VisibilityTester *visibility) {
    *wi = Normalize(lightPos - p);
    visibility->SetSegment(p, lightPos);
    return Intensity * Falloff(-*wi)
        /DistanceSquared(lightPos,p);
}
```

Spotlights

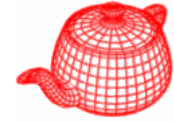


```
float SpotLight::Falloff(const Vector &w) const {
    Vector wl = Normalize(WorldToLight(w));
    float costheta = wl.z;
    if (costheta < cosTotalWidth)
        return 0.;
    if (costheta > cosFalloffStart)
        return 1.;
    float delta = (costheta - cosTotalWidth) /
                  (cosFalloffStart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

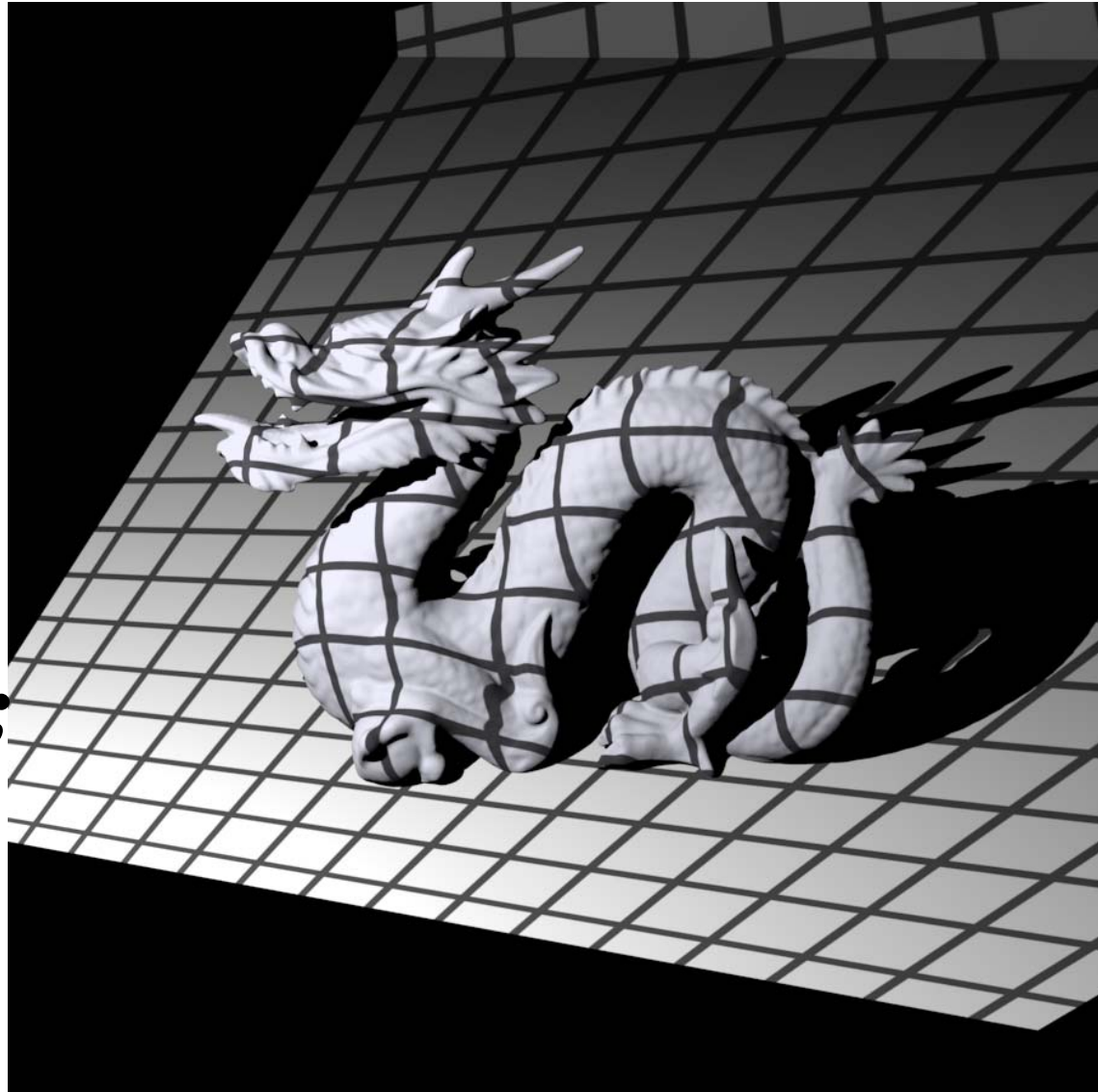
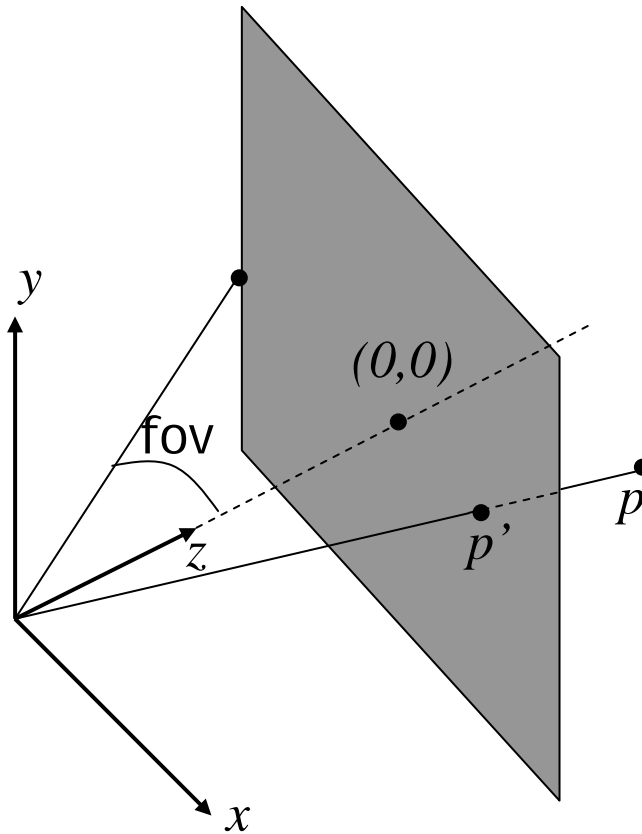
$$\int_{\Omega'} d\omega = \int_{\phi=0}^{2\pi} \int_{\theta=0}^{\theta'} \sin \theta d\theta d\phi = \int_{\phi=0}^{2\pi} (1 - \cos \theta') d\phi = 2\pi(1 - \cos \theta')$$

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

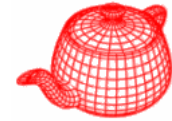
Texture projection lights



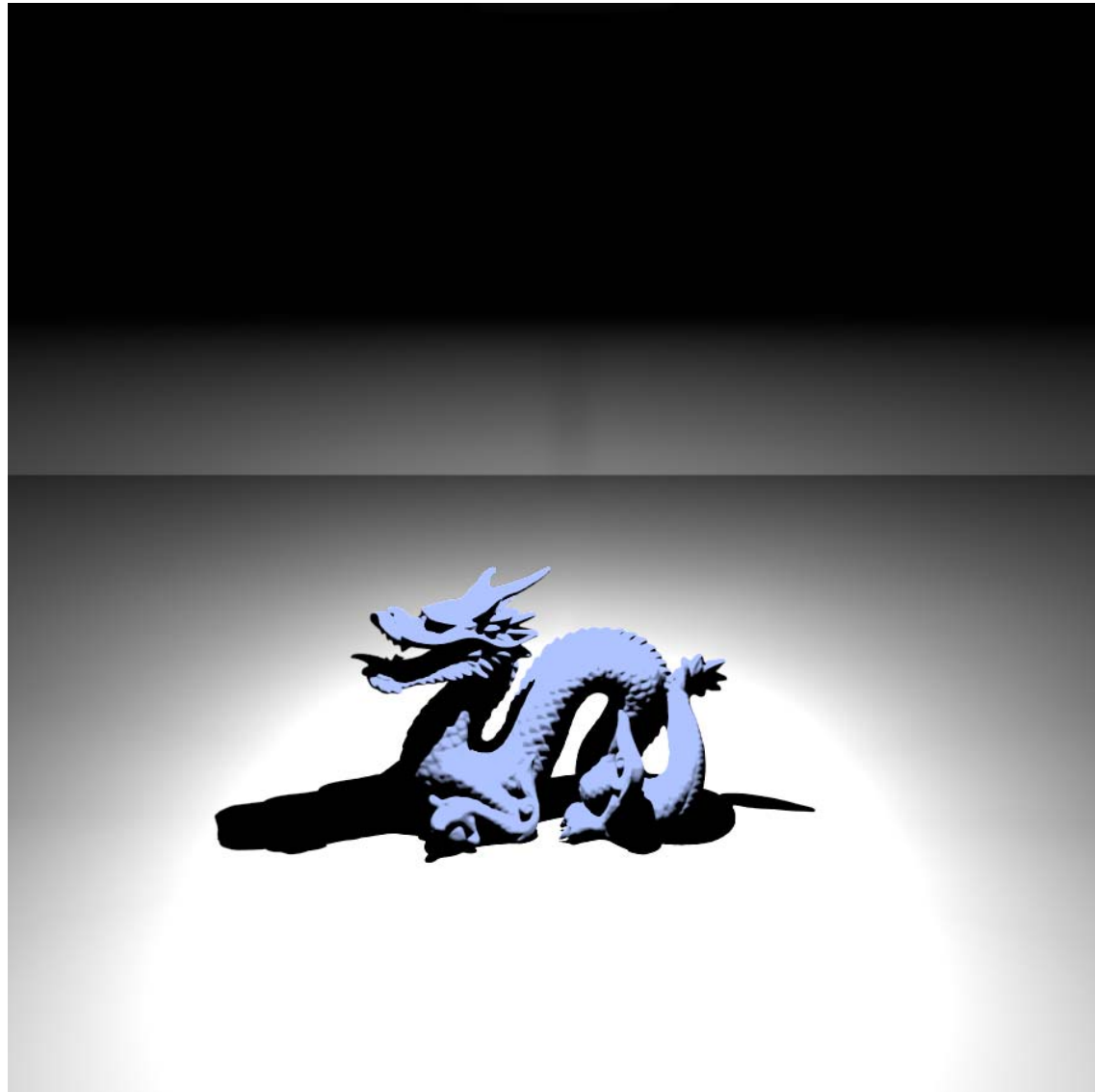
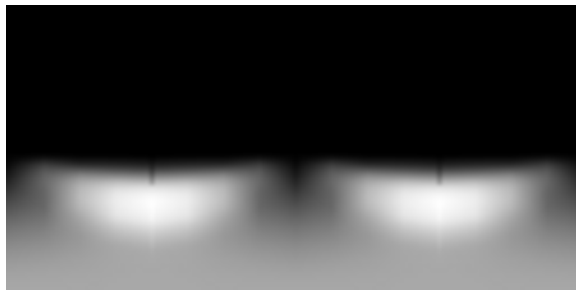
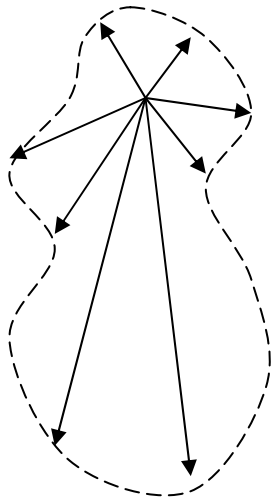
- Like a slide projector



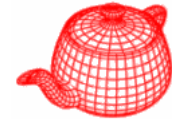
Goniophotometric light



- Define a angular distribution from a point light



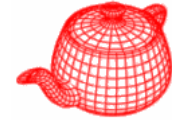
Goniophotometric light



```
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, Vector *wi,
    VisibilityTester *visibility) const {
    *wi = Normalize(lightPos - p);
    visibility->SetSegment(p, lightPos);
    return Intensity * Scale(-*wi)
        / DistanceSquared(lightPos, p);
}
```

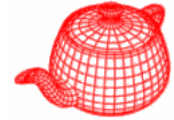
Goniophotometric light



```
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_TWOPHI, t = theta * INV_PI;  
    return mipmap ? mipmap->Lookup(s, t) : 1.f;  
}
```

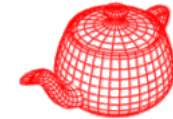
```
Spectrum Power(const Scene *) const {  
    return 4.f * M_PI * Intensity *  
           mipmap->Lookup(.5f, .5f, .5f);  
}
```

Point lights



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

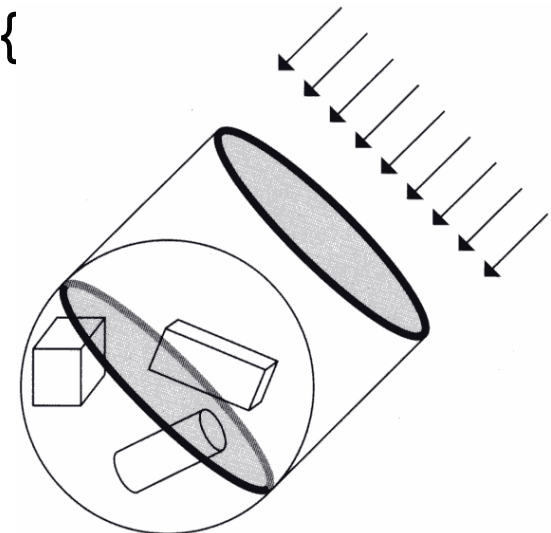
Directional lights



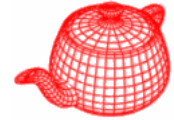
```
DistantLight::DistantLight(Transform &light2world,  
Spectrum &radiance, Vector &dir):Light(light2world) {  
    lightDir = Normalize(LightToWorld(dir));  
    L = radiance;  
}
```

```
Spectrum DistantLight::Sample_L(Point &p, Vector *wi,  
VisibilityTester *visibility) const {  
    wi = lightDir;  
    visibility->SetRay(p, *wi);  
    return L;  
}
```

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



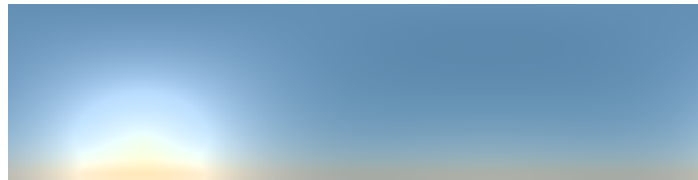
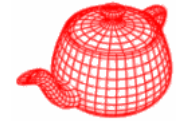
Area light



- Defined by a **shape**
- Uniform over the surface
- Single-sided
- **sample_L** isn't straightforward because a point could have contributions from multiple directions (chap15).



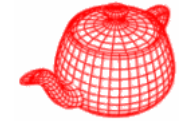
Infinite area light



morning skylight



Infinite area light



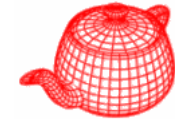
midday skylight



sunset skylight



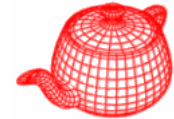
Infinite area light



```
InfiniteAreaLight(Transform &light2world, Spectrum &L,
    int ns, string &texmap) : Light(light2world, ns) {
    radianceMap = NULL;
    if (texmap != "") {
        int w, h;
        Spectrum *texels = ReadImage(texmap, &w, &h);
        if (texels) radianceMap =
            new MIPMap<Spectrum>(w, h, texels);
        delete[] texels;
    }
    Lbase = L;
}

Spectrum Power(const Scene *scene) const {
    Point wldC; float wldR;
    scene->WorldBound().BoundingSphere(&wldC, &wldR);
    return Lbase * radianceMap->Lookup(.5f, .5f, .5f)
        * M_PI * wldR * wldR;
}
```

Infinite area light



```
Spectrum Le(const RayDifferential &r) {  
    Vector w = r.d;           for those rays which miss the scene  
    Spectrum L = Lbase;  
    if (radianceMap != NULL) {  
        Vector wh = Normalize(WorldToLight(w));  
        float s = SphericalPhi(wh) * INV_TWOPHI;  
        float t = SphericalTheta(wh) * INV_PI;  
        L *= radianceMap->Lookup(s, t);  
    }  
    return L;  
}
```