Assignment 7: Path Tracing

Mengzhu Wang

March 2, 2024

1 Radiometry

1.1 Radiant energy

Radiant energy is the energy of electromagnetic radiation. It is measured in units of joules, and denoted by the symbol

$$Q[J = Joule]$$

1.2 Radiance flux/power

Radiant flux (power) is the energy emmitted, reflected, transmitted or received, per unit time.

$$\Phi = \frac{dQ}{dt}[w = Watt][lm = lumen]$$

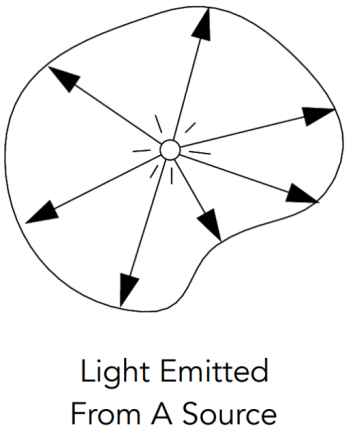
1.3 Radiant intensity

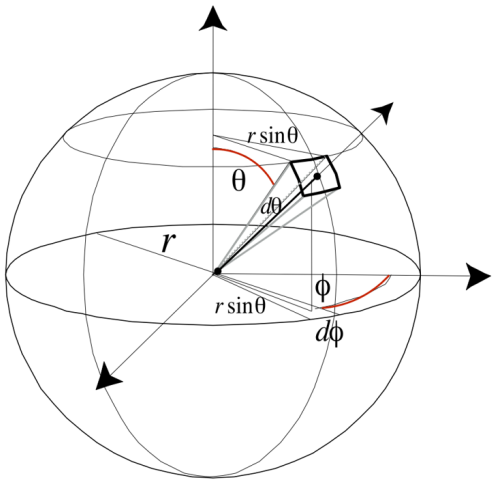
The radiant (luminous) intensity is the power per unit solid angle emitted by a point light source.

$$I(\omega) = \frac{d\Phi}{d\omega}[\frac{W}{sr}][\frac{lm}{sr} = cd = candela]$$

Solid angle is the ratio of subtended area on sphere to radius squared $\Omega = A/r^2$. A tiny solid angle is related to a tiny area on the surface, which can be calculated by $dA = (rd\theta)(r\sin\theta d\phi) = r^2\sin\theta d\theta d\phi$. Differential solid angle is $d\omega = dA/r^2 = \sin\theta d\theta d\phi$. Solid angle of entire sphere is $\Omega = \int_{S^2} d\omega = \int_0^{2\pi}\int_0^{\pi}\sin\theta d\theta d\phi = 4\pi$.

For isotropic point source, $\Phi = \int_{S^2} Id\omega = 4\pi I$. Therefore, $I = \Phi/4\pi$.



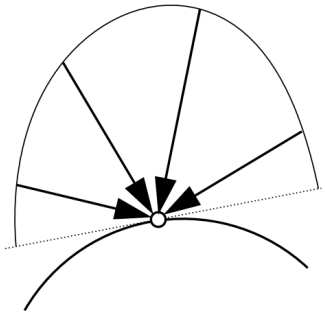


1.4 Irradiance

The irradiance is the power per unit area incident on a surface point.

$$E(x) = \frac{d\Phi(x)}{dA}[\frac{W}{m^2}][\frac{lm}{m^2} = lux]$$

According to Lambert's Cosine Law, irradiance at surface is proportional to cosine of angle between light direction and surface normal $E = \frac{\Phi}{A}\cos\theta$. Irradiance also has falloff with sphere radius increases by $E' = \frac{\Phi}{4\pi r^2} = \frac{E}{r^2}$.

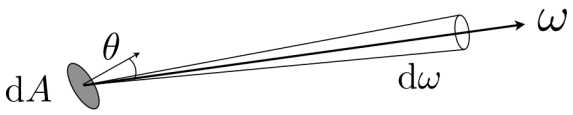


Light Falling
On A Surface

1.5 Radiance

The radiance (luminance) is the power emitted, reflected, transmitted or received by a surface, per unit solid angle, per projected unit area. $\cos\theta$ accounts for projected surface area.

$$L(p,\omega) = \frac{d^2\Phi(p,\omega)}{d\omega\ dA\cos\theta}[\frac{W}{sr\ m^2}][\frac{cd}{m^2} = \frac{lm}{sr\ m^2} = nit]$$



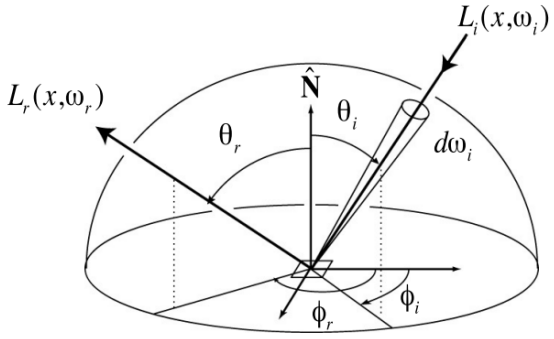
1.6 Irradiance vs. Radiance

- Irradiance: total power received by area dA
- Radiance: power received by area dA from direction $d\omega$

$$dE(p,\omega) = L(p,\omega)(n\cdot\omega)d\omega$$
$$E(p) = \int_{\Omega^+} L_i(p,\omega_i)(n\cdot\omega_i)d\omega_i$$

2 Bidirectional Reflectance Distribution Function (BRDF)

Radiance from direction ω_i turns into the power E that dA receives. Then power E will become the radiance to any other direction ω_o. Differential irradiance incoming is $dE(p,\omega_i) = L(p,w_i)(n\cdot\omega_i)d\omega_i$,



and differential radiance exiting is $dL(p, \omega_o)$. The BRDF represents how much light is reflected into each outgoing direction ω_o from each incoming direction,

$$f_r(\omega_i \to \omega_o) = \frac{dL(p, \omega_o)}{dE(p, \omega_i)} = \frac{dL(p, \omega_o)}{L(p, \omega_i)(n \cdot \omega_i)d\omega_i}[\frac{1}{sr}]$$

The reflection equation is

$$L_o(p, \omega_o) = \int_{\Omega^+} f_r(p, \omega_i \to \omega_o) L_i(p, \omega_i)(n \cdot \omega_i)d\omega_i$$

Rewrite the reflection equation by adding an emission term to make it general, we can get the rendering equation

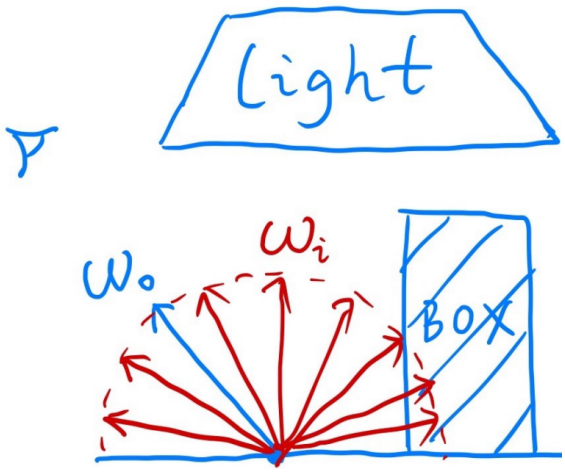
$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{\Omega^+} f_r(p, \omega_i, \omega_o) L_i(p, \omega_i)(n \cdot \omega_i)d\omega_i$$

3 Monte Carlo Integration

Estimate the integral of a function by averaging random samples of the function's value.

$$\int f(x)dx = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}, X_i \sim p(x)$$

4 Path Tracing

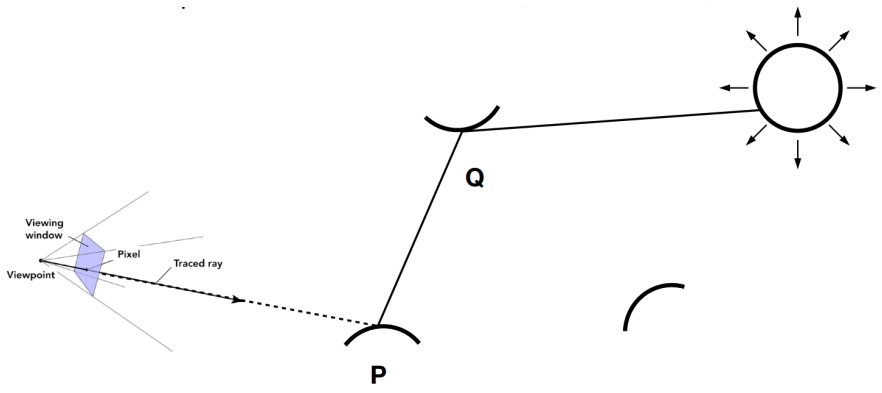


Render one pixel (point) for direct illumination only, we need to compute the radiance at p towards the camera.

$$L_o(p, \omega_o) = \int_{\Omega^+} f_r(p, \omega_i, \omega_o) L_i(p, \omega_i)(n \cdot \omega_i)d\omega_i$$

According to Monte Carlo Integration, we have $f(x) = f_r(p, \omega_i, \omega_o)L_i(p, \omega_i)(n \cdot \omega_i)$ and pdf is $p(\omega_i) = 1/2\pi$ assuming uniformly sampling the hemisphere. So, in general,

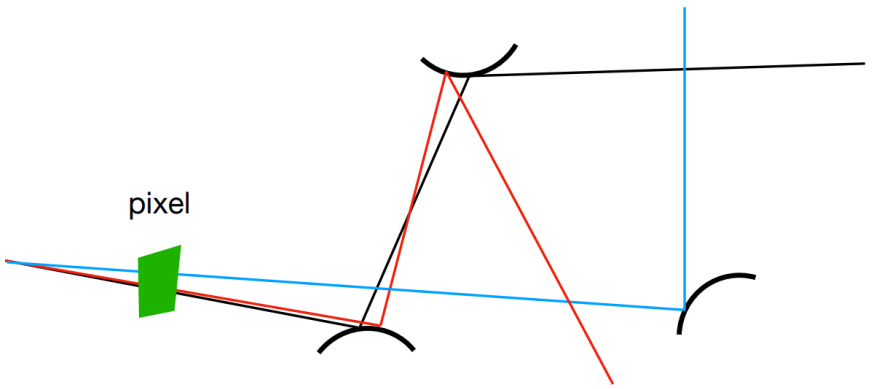
$$L_o(p, \omega_o) = \int_{\Omega^+} f_r(p, \omega_i, \omega_o)L_i(p, \omega_i)(n \cdot \omega_i)d\omega_i \approx \frac{1}{N}\sum_{i=1}^{N} \frac{f_r(p, \omega_i, \omega_o)L_i(p, \omega_i)(n \cdot \omega_i)}{p(\omega_i)}$$



If a ray hits an object, for example, Q also reflects light to P, we need to introduce global illumination as pseudocode below.

```
shade(p, wo)
    Randomly choose N directions wi~pdf
    Lo = 0.0
    For each wi
        Trace a ray r(p, wi)
        If ray r hit the light
            Lo += (1 / N) * L_i * f_r * cosine / pdf(wi)
        Else If ray r hit an object at q
            Lo += (1 / N) * shade(q, -wi) * f_r * cosine
            / pdf(wi)
    Return Lo
```

However, there is a problem that rays explode as boounces go up. Only 1 ray is traced at each shading point is ok but this will be noisy. We can just trace more paths through each pixel and average their radiance.



```
ray_generation(camPos, pixel)
    Uniformly choose N sample positions within the pixel
    pixel_radiance = 0.0
    For each sample in the pixel
        Shoot a ray r(camPos, cam_to_sample)
        If ray r hit the scene at p
            pixel_radiance += 1 / N * shade(p, sample_to_cam)
    Return pixel_radiance
shade(p, wo)
    Randomly choose ONE direction wi~pdf(w)
    Trace a ray r(p, wi)
    If ray r hit the light
        Return L_i * f_r * cosine / pdf(wi)
    Else If ray r hit an object at q
        Return shade(q, -wi) * f_r * cosine / pdf(wi)
```

The second problem is the recursive algorithm in shade() will never stop. Our solution is Russian Roulette (RR). Previously, we always shoot a ray at a shading point and get the shading result L_o. Suppose we manually set a probability P (0<P<1). With probability P, shoot a ray and return the shading result divided by P, which is L_o/P. With probability 1-P, don't shoot a ray and we'll get 0. In this way, we can still expect to get L_o because $E = P * (L_o/P) + (1 - P) * 0 = L_o$.

```
shade(p, wo)
    Manually specify a probability P_RR
    Randomly select ksi in a uniform dist. in [0, 1]
    If (ksi > P_RR) return 0.0;

    Randomly choose ONE direction wi~pdf(w)
    Trace a ray r(p, wi)
    If ray r hit the light
        Return L_i * f_r * cosine / pdf(wi) / P_RR
    Else If ray r hit an object at q
        Return shade(q, -wi) * f_r * cosine / pdf(wi) / P_RR
```

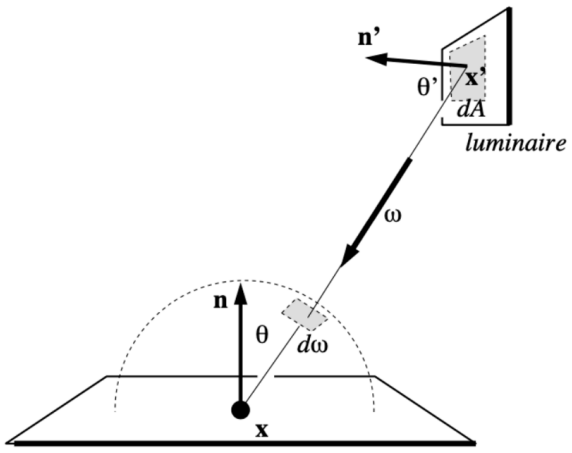
Although now we already have a correct version of path tracing, it's not really efficient. With low SPP (samples per pixel), we will get noisy results. The smaller the light, the lower the probability that the rays hit the light. So a lot of rays are wasted if we uniformly sample the hemisphere at the shading point.

We can sample the light therefore no rays are wasted. We need to make the rendering equation as an integral of dA which requires the relationship between $d\omega$ and dA. The alternative definition of solid angle is projected area on the unit sphere,

$$d\omega = \frac{dA \ \cos\theta'}{\|x' - x\|^2}$$

We can rewrite the rendering equation as

$$L_o(p, \omega_o) = \int_{\Omega^+} f_r(p, \omega_i, \omega_o) L_i(p, \omega_i)(n \cdot \omega_i) d\omega_i = \int_{\Omega^+} f_r(p, \omega_i, \omega_o) L_i(p, \omega_i) \frac{(n \cdot \omega_i) * (n' \cdot \omega_i)}{\|x' - x\|^2} dA$$



As for Monte Carlo integration, f(x) is everything inside and pdf is $1/A$. Now we consider the radiance coming from two parts:

- light source (direct, no need to have RR)
- other reflectors (indirect, RR)

In addition, we should judge if the sample on the light is not blocked or not.

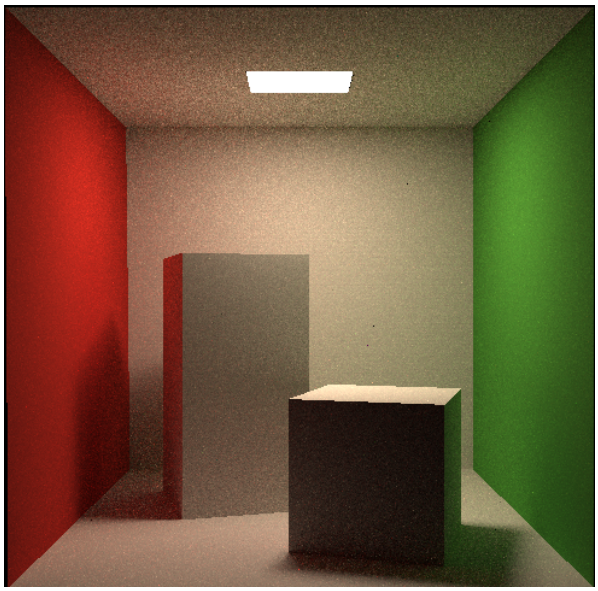
```
shade(p, wo)
  Uniformly sample the light at xx (pdf_light = 1 / A)
  Shoot a ray from p to x
  If the ray is not blocked in the middle
    L_dir = L_i * f_r * cos_theta * cos_theta_x / |x-p|^2
  / pdf_light

  L_indir = 0.0
  Test Russian Roulette with probability P_RR
  Uniformly sample the hemisphere toward wi (pdf_hemi = 1
  / 2pi)
  Trace a ray r(p, wi)
  If ray r hit a non-emitting object at q
    L_indir = shade(q, wi) * f_r * cos_theta / pdf_hemi /
  P_RR

  Return L_dir + L_indir
```

5 Multithreading

All pixels generate primary rays and cast these rays into the scene, so we can divide threads by rows. Lambda method is used to create the function called by each thread. To avoid resource contention, we select mutex.

Plus, function get_random_float() costs a lot. Add static to three variables can avoid duplicate builds, thus accelerating the rendering. The results when SPP is 128 show below,

- w/o multi-thread Time = 805 seconds
- w/ multi-thread Time = 145 seconds



6 Microfacet