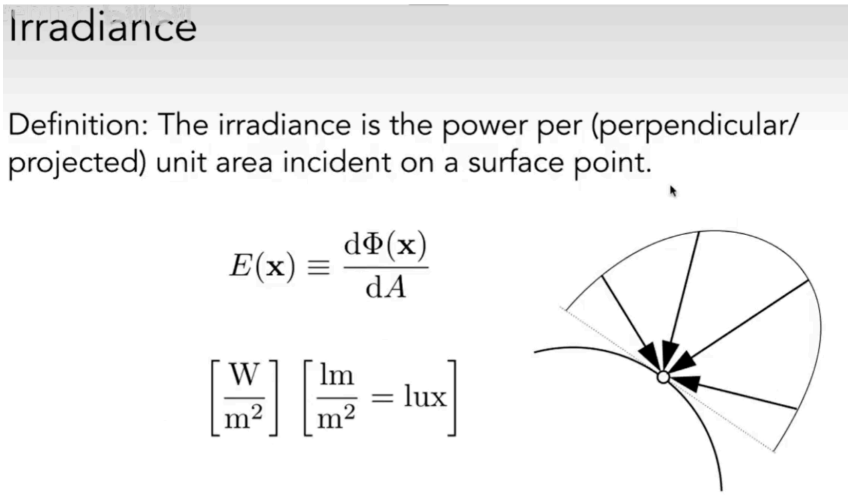
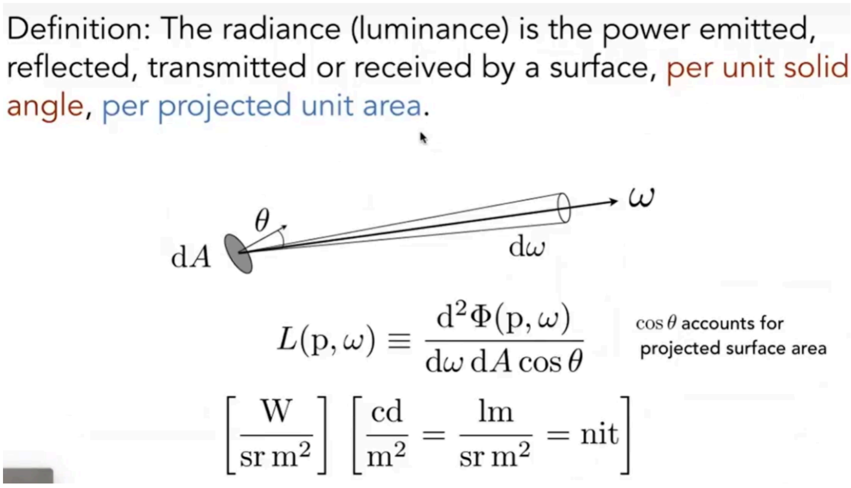
首先是之前的 radiant flux 和 radiant intensity



irradiance 更适合翻译成 **面积辐照能率**，因为是每单位面积每秒，两个计量轴。辐照在这里是接受辐射的意思。

这里提到了只有垂直于表面的才算打上去的光。

radiant intensity 不会随距离衰减，irradianace 才会。



radiance，嗯，可以翻译成 **角面积辐射能率**，也可以翻译成 **角面积辐照能率**，因为它既可以描述辐射，也可以描述接收辐射，或者传输等，那就是 **角面积辐射通量**。

这个感觉应该和 flux 跟 intensity 一起讲。好吧老师真一起讲了，radiance 既可以是 irradiance per unit solid angle，也可以是 intensity per unit projected area.

radiance 相比 intensity 的优点是它可以描述体积光源。

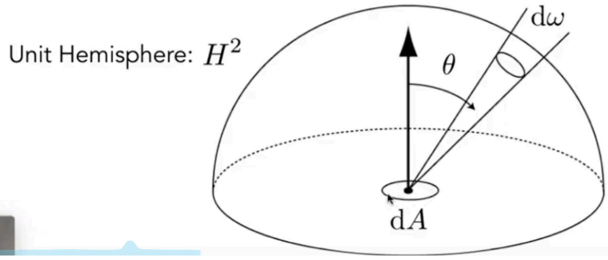
哦豁，nit，现在手机圈搞噱头都用这个单位。

Irradiance: total power received by area dA

Radiance: power received by area dA from “direction” dω

$$dE(\mathrm{p},\omega) = L_i(\mathrm{p},\omega)\cos\theta\,\mathrm{d}\omega$$

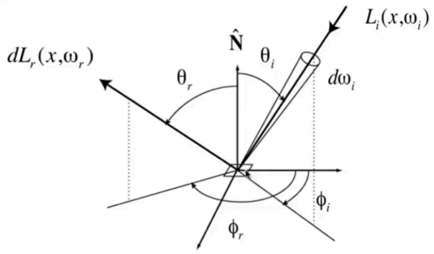
$$E(\mathrm{p}) = \int_{H^2} L_i(\mathrm{p},\omega)\cos\theta\,\mathrm{d}\omega$$



嗯，写了个积分。

BRDF(Bidirectional Reflectance Distribution Fuction)双向反射分布函数

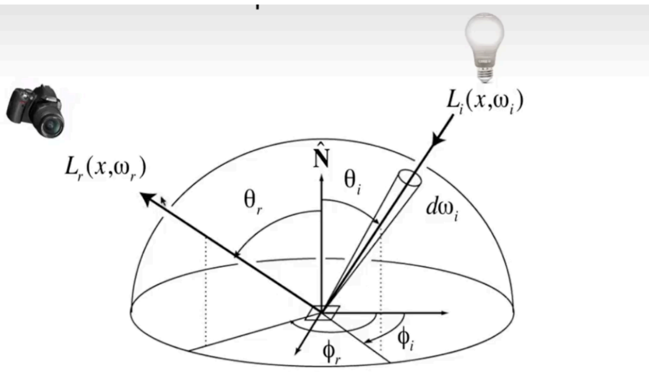
The Bidirectional Reflectance Distribution Function (BRDF) represents how much light is reflected into each outgoing direction ω_r from each incoming direction



$$f_r(\omega_i \to \omega_r) = \frac{\mathrm{d}L_r(\omega_r)}{\mathrm{d}E_i(\omega_i)} = \frac{\mathrm{d}L_r(\omega_r)}{L_i(\omega_i)\cos\theta_i\,\mathrm{d}\omega_i} \quad \left[\frac{1}{\mathrm{sr}}\right]$$

他这个写的有点乱，直接理解成入射方向给这个单位面积提供的 irradiance 给出射方向提供了多少 radiance 就行了。或者看公式它写的是个比率，也就是 radiance per irradiance，这个东西其实就是在定义某种材质。

用公式理解可能快一些。



$$L_r(\mathrm{p},\omega_r) = \int_{H^2} f_r(\mathrm{p},\omega_i \to \omega_r)\,L_i(\mathrm{p},\omega_i)\,\cos\theta_i\,\mathrm{d}\omega_i$$

Rendering Equation 渲染方程

The Rendering Equation

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

Note: now, we assume that all directions are pointing outwards!

其实就是把自发光也考虑进去

嗯，那么怎么解渲染方程呢，毕竟这是个比较递归的定义，因为自己发出的光也可以经过反射后回来。

答案是……我去，生成函数！

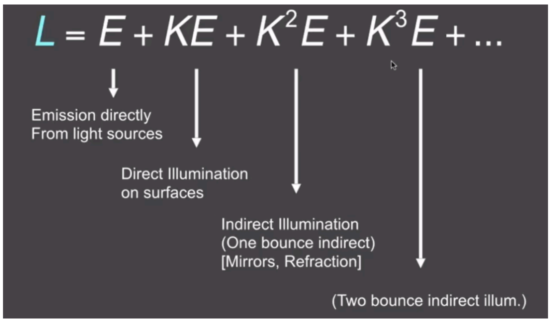
- Approximate set of all paths of light in scene

$$L = E + KL$$
$$IL - KL = E$$
$$(I-K)L = E$$
$$L = (I-K)^{-1}E$$

Binomial Theorem

$$L = (I + K + K^2 + K^3 + \ldots)E$$
$$L = E + KE + K^2E + K^3E + \ldots$$

看来这就是光线追踪软件里开几级光线追踪的来源了，后面讲到光栅化就是 $E+KE$ 这部分。



提到了光线追踪的收敛，嗯，看来要学无穷级数了。

什么，是概率论！直接跳到连续情况吧。

Conditions on p(x): $p(x) \geq 0 \text{ and } \int p(x)\,dx = 1$

Expected value of X: $E[X] = \int x\,p(x)\,dx$

这里注意 PDF(Probability Distribution Function) 的某个值并不是概率，它的一段定积分才是概率。

芜，还有？

Expected value of a function of a random variable:

$$E[Y] = E[f(X)] = \int f(x)\, p(x)\, dx$$

吔，是复合函数期望法则，不过挺好理解的

Monte Carlo Integration

一种数值方法，可以在不求不定积分的前提下对函数做定积分。

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

Some notes:

- The more samples, the less variance.
- Sample on x, integrate on x.

说一下这个的原理。

$$\int_a^b f(x)dx = \int_a^b p(x)\frac{f(x)}{p(x)}dx = E\left[X = \frac{f(x)}{p(x)}\right]$$

E 是在 $p(x)$ 下成立的。注意，到这一步其实还是在做确定性的东西，下面才是估计的部分。

蒙特卡洛积分真正估计的是 $E[X]$.

在 $E[X]$ 的约束（也即 $p(x)$ 作为 PDF）下采样 N 次求平均，就是估计过程了，很暴力。

Path Tracing

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

```
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)
    Return Lo
```

嗯，确实很简单。

接下来是添加全局光照的支持。这样看来路径追踪的允许弹射次数就相当于这个 shade 函数的最大递归深度。(实际弹射次数=递归深度-1，但不要过于在意这类数值边界)

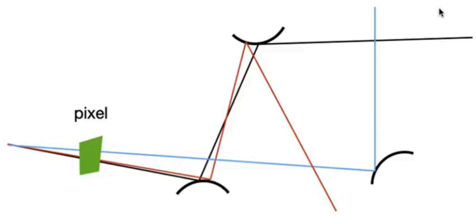
```
shade(p, wo)
    Randomly choose N directions wi~pdf
    Lo = 0.0
    For each wi
        Trace a ray r(p, wi)
        If ray r hit the light
            Lo += (1 / N) * L_i * f_r * cosine / pdf(wi)
        Else If ray r hit an object at q
            Lo += (1 / N) * shade(q, -wi) * f_r * cosine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

嗯，然后立马就讲了这个方法还是会复杂度爆炸。然后提到只反射一根光线不会指数爆炸，就是路径追踪了。

N = 1 的采样率实在太小，Monte Carlo 积分方法噪声会非常大，提到的解决方式是一个像素发出多道光线，虽然不会增加蒙特卡洛计算的采样率，但增加了这一个像素的采样率，算是一种全局性的采样补偿。

But this will be noisy!

No problem, just trace more **paths** through each pixel and average their radiance!



又提到了除了递归深度之外方法来逼近现实的弹射次数效果：俄罗斯轮盘赌。

也即用一个概率来决定递归是否向下，当然具体计算的时候要保证得到的值的期望等同于无限弹射的场景。这也是这个方法的精妙之处。

Suppose we manually set a probability P (0 < P < 1)

With probability P, shoot a ray and
return the **shading result divided by P**: **Lo / P**

With probability 1-P, don't shoot a ray and you'll get **0**

In this way, you can still **expect** to get Lo!:

E = P * **(Lo / P)** + (1 - P) * **0** = **Lo**

嗯哼，做一下课上提到的简单题。

期望弹射的次数其实就是 $(1-p)\sum_{i\geq 0} p^i i$，后面是个收敛的无穷级数，算出来是 $\frac{p}{(1-p)^2}$，所以答案是 $\frac{p}{1-p}$

修改 shade 算法其实有两个方向，一个是让概率分布函数更倾向于光源，另一个就是课上说的这种了。

Need to make the rendering equation as an integral of dA

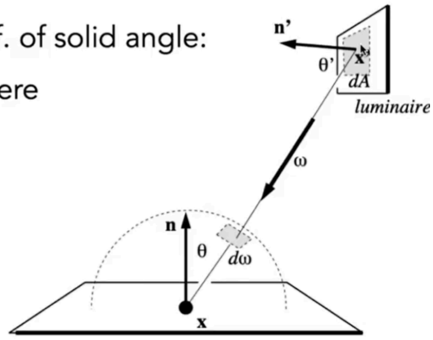
Need the relationship between dω and dA

Easy! Recall the alternative def. of solid angle:

Projected area on the unit sphere

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

(Note: θ', not θ)




```
shade(p, wo)
    # Contribution from the light source.
    Uniformly sample the light at x' (pdf_light = 1 / A)
    L_dir = L_i * f_r * cos θ * cos θ' / |x' - p|^2 / pdf_light

    # Contribution from other reflectors.
    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 θ / pdf_hemi / P_RR

    Return L_dir + L_indir
```

看代码还是更直观，这里只对光源进行直接采样，对反射过来的光还是原先的采样方法。

Path-Tracing 真正做到了 Photo-Realistic.

老师提到了概念区分，可以。

以前说光线追踪更多指的是 Whitted-Styled 光线追踪，现在则更像一种统称。

又提到了一些话题，嗯，听的我起身独立向荒原了。

做一下作业 6.

首先是适配一下 castRay 的新接口，新框架封装了一个 Ray.hpp

```
Vector3f dir = normalize(Vector3f(x, y, -1)); // Don't forget to normalize this direction!
Ray newray(eye_pos, dir);
framebuffer[m++] = scene.castRay(newray, 0);
```

然后嘛，由于新框架的注释给的太少，你必须深入翻一下它的接口才能明白后面的题到底想让你干啥。而且它这个框架真的是挺想让人吐槽的……

比如从 Triangle::getIntersection 开始吧，你会发现有一个叫做 rayTriangleIntersect 的函数，实现的是几乎一样的功能……

当然重点是一个叫 Intersection 的作为返回值的结构体。于是发现了一个叫 Object 的类，他是干啥的？后来知道三角形类就是继承的 Object，这才反映过来是 C++ 的面向对象。

于是添加了如下代码。

```
if (t_tmp < EPSILON) // 这里有些武断，但不会有大问题
        return inter;

    inter.happened = true;
    inter.coords = u * v0 + v * v1 + (1 - u - v) * v2;// 也可以写成 ray(t_tmp)
    inter.distance = t_tmp;
    inter.normal = normal;
    inter.obj = this;
    inter.m = m;
```

接下来是 Bounds 的内容。你得知道 pMin 和 pMax 是三对数对而不是两个向量……嗯……我当成了两个向量所以把三对平面直接误会成了一对平面，很难蹦。

```
void myswap(float &x, float &y) {
    float tmp = x;
    x = y;
    y = tmp;
}

inline bool Bounds3::IntersectP(const Ray& ray, const Vector3f& invDir,
                                const std::array<int, 3>& dirIsNeg) const
{
    // invDir: ray direction(x,y,z), invDir=(1.0/x,1.0/y,1.0/z), use this because
Multiply is faster that Division
    // dirIsNeg: ray direction(x,y,z), dirIsNeg=[int(x>0),int(y>0),int(z>0)], use
this to simplify your logic
    // TODO test if ray bound intersects
    float tMin_x = (pMin.x - ray.origin.x) * invDir.x;
    float tMax_x = (pMax.x - ray.origin.x) * invDir.x;
    float tMin_y = (pMin.y - ray.origin.y) * invDir.y;
    float tMax_y = (pMax.y - ray.origin.y) * invDir.y;
    float tMin_z = (pMin.z - ray.origin.z) * invDir.z;
    float tMax_z = (pMax.z - ray.origin.z) * invDir.z;

    if(!dirIsNeg[0]) myswap(tMin_x, tMax_x);
    if(!dirIsNeg[1]) myswap(tMin_y, tMax_y);
    if(!dirIsNeg[2]) myswap(tMin_z, tMax_z);
    // 这里为什么要翻转呢，因为总是有 pMin.x < pMax.x，但我们的光线方向不一定从小到大
    // 就导致你会把后相交的那个 t 当成 tMin，先相交的那个 t 当成 tMax

    float tMin = std::fmax(tMin_x, std::fmax(tMin_y, tMin_z)),
          tMax = std::fmin(tMax_x, std::fmin(tMax_y, tMax_z));

    if(tMin < tMax && tMax >= EPSILON) return true;
    return false;

}
```

最后的 build_bvh 反而是最简单的，因为理解了 Object 类和 C++ 的继承……

```
Intersection BVHAccel::getIntersection(BVHBuildNode* node, const Ray& ray) const
{
    // TODO Traverse the BVH to find intersection
    Intersection isect;
    std::array<int, 3> dirIsNeg;
    dirIsNeg[0] = ray.direction.x > 0;
    dirIsNeg[1] = ray.direction.y > 0;
    dirIsNeg[2] = ray.direction.z > 0;


    if(!node->bounds.IntersectP(ray, ray.direction_inv, dirIsNeg)) return isect;
    if(node->left == nullptr || node->right == nullptr) return node->object-
>getIntersection(ray);
    Intersection lsect = getIntersection(node->left, ray), rsect =
getIntersection(node->right, ray);
    return lsect.distance < rsect.distance ? lsect : rsect;

}
```