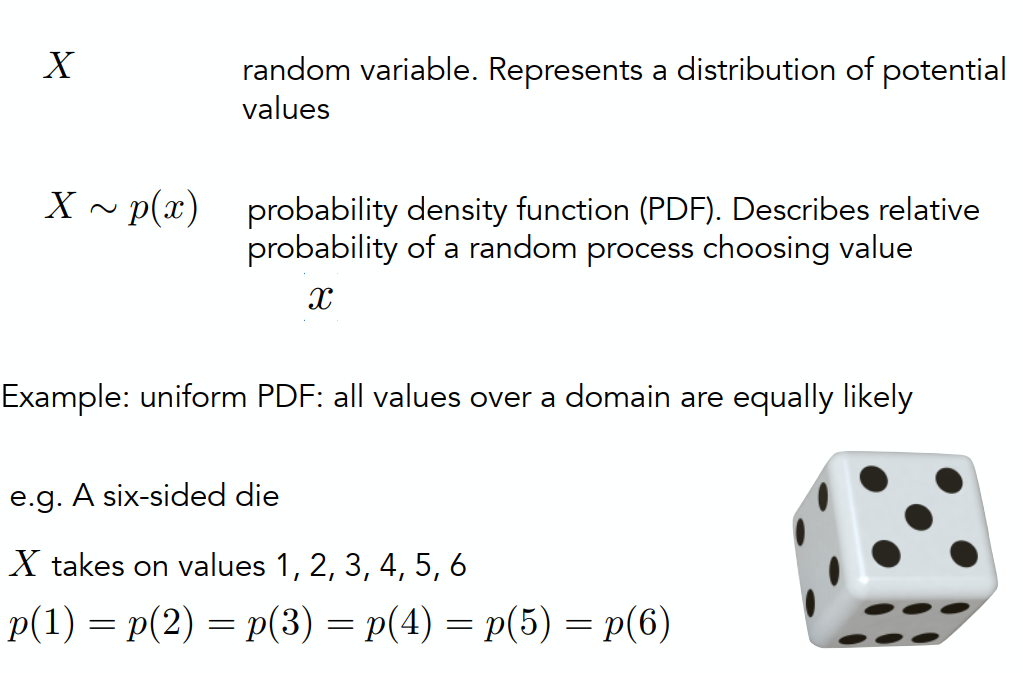
# 一、概率复习

## 1、随机变量

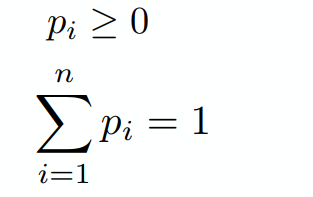
X：随机变量，表示潜在值的分布

X~p(x)：概率密度函数 (PDF)，描述随机过程选择值的相对概率

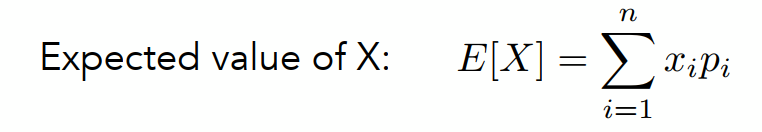
示例：均匀分布：一个域中的所有值都具有相同的可能性(掷骰子掷到1-6的可能性皆为1/6)

## 2、随机变量的期望值

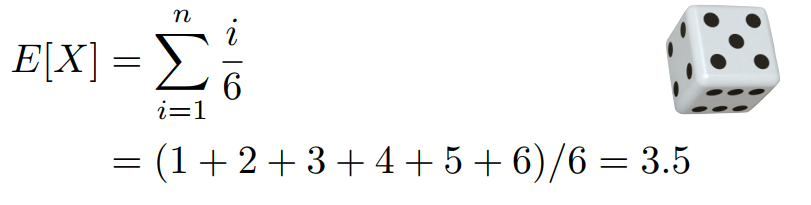
n个离散值：xi 概率：pi



X 的期望值：



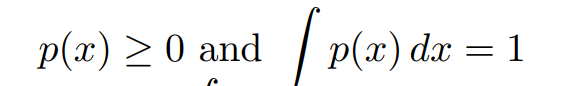
计算骰子点数的期望：



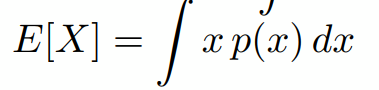
## 3、连续情况：概率分布函数 (PDF)

概率密度函数X~p(x)：一个随机变量 X，它可以取一组连续值中的任何一个，其中特定值的相对概率由连续概率密度函数 p(x) 给出。

条件:

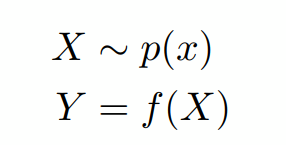


期望:

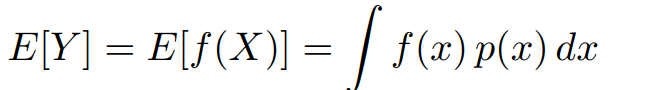


概率密度函数所有可能的取值（面积）为1

随机变量X的函数Y也是随机变量，即：



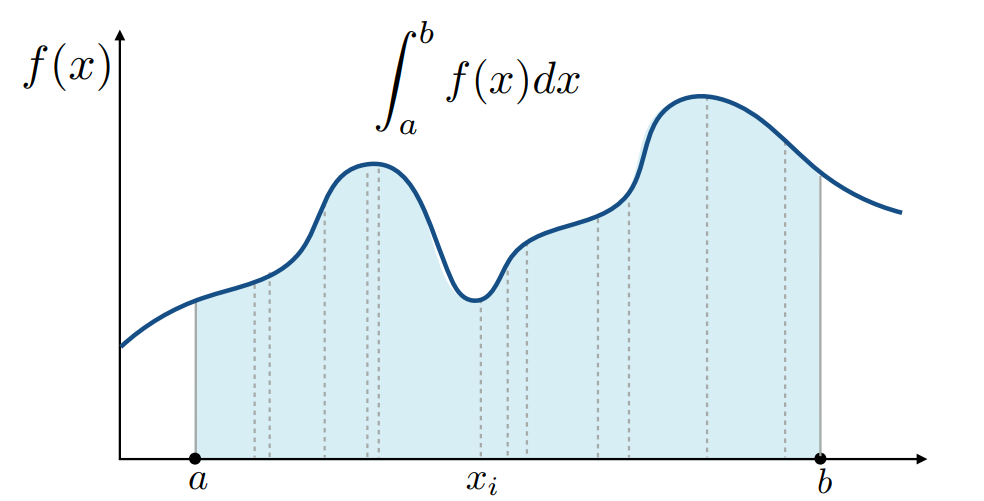
Y的期望为：



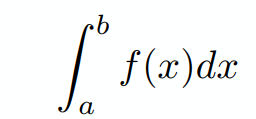
# 二、蒙特卡洛积分

当我们想求一个函数的积分值，但是由于函数太复杂(如下)，无法写出解析式，从而无法求出积分值的时候，我们可以通过蒙特卡洛积分来求。

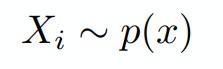
蒙特卡洛积分是一种数值方法，只考虑最后经采样得到的随机变量的面积和与原始积分近似。



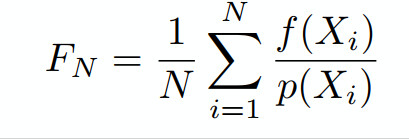
若原始积分定义为：



Xi是[a,b]的x取值，它满足

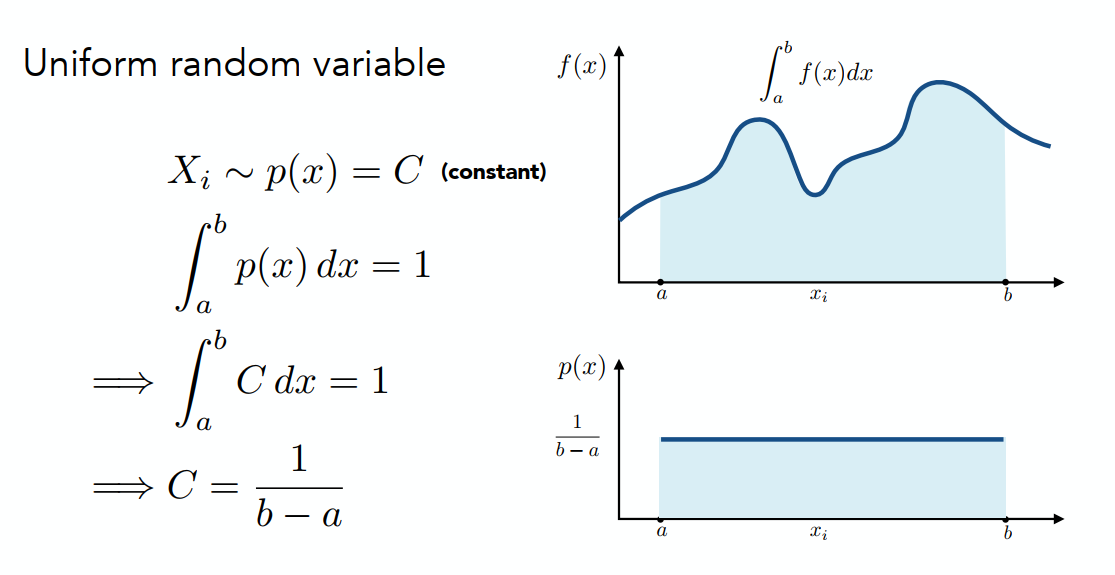


蒙特卡洛积分定义为：

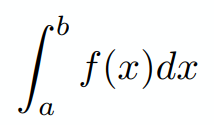


通过一个例子来解释上述公式：

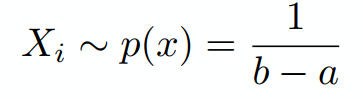
如果我想在a到b之间均匀采样，那么就说明采样中我们用的概率密度函数(PDF)应该是一个常数，写作C。由于PDF在积分域上积分出来的结果是1，那么PDF解出来就是1/b-a。此处也可以这样理解：PDF在a到b上积分是1，又是个常数，所以图像应该是个矩形，我们知道这个矩形的底是b-a，面积又得是1，那么高就是1/b-a。



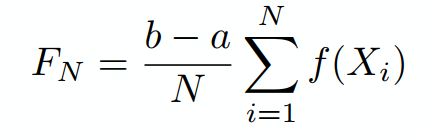
那么用蒙特卡洛积分的方法去算原始积分：



已知PDF：



则蒙特卡洛积分为：



这也就是蒙特卡洛积分的一种特殊情况，即PDF是均匀的。

# 三、Path Tracing路径追踪

## 1、Whitted-Style Ray Tracing: Problem

Whitted-style ray tracing对于光线的弹射方法依据以下两个原则：

当光线打到光滑物体上，会沿着镜面方向反射或沿折射方向折射

当光线打到漫反射物体，那么这条光线就停下来了

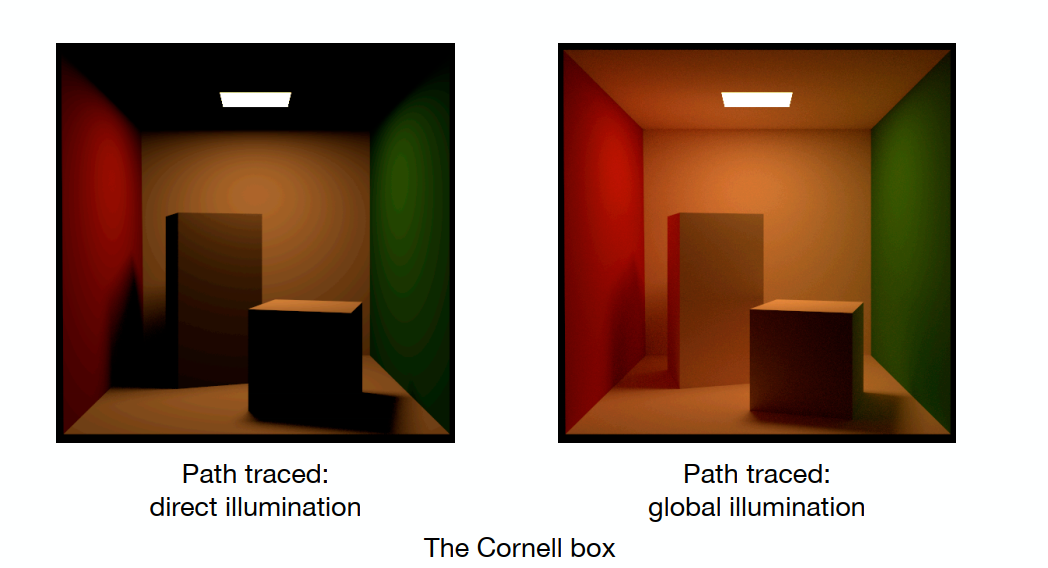
Whitted-style ray tracing存在以下两个问题：

### Problem1



这两个茶壶，左边这个完全光滑，是镜面反射，右边这个相对来讲没有那么光滑。在Whitted-Style Ray Tracing下，对于镜面反射的物体来讲没有什么问题，但是对于表面有些粗糙的物体，反射的光线应该朝向各个方向，如果认为还是认为沿着镜面反射方向反射，是不对的，因此这里产生了问题。

### Problem2



上面这个场景是用路径追踪得到的，场景中的物体都是漫反射的，而如果用Whitted-style ray tracing，其定义当光线打到漫反射物体，那么这条光线就停下来了，那么两个物体之间的光线全部都得不到。

因此，Whitted-style ray tracing存在不足，我们需要提出更高级的方法来解释更贴近现实的场景：

虽然Whitted-Style Ray Tracing是错的，但是渲染方程是对的。为了解这个渲染方程，考虑

①光是来自四面八方的，要考虑对半球积分

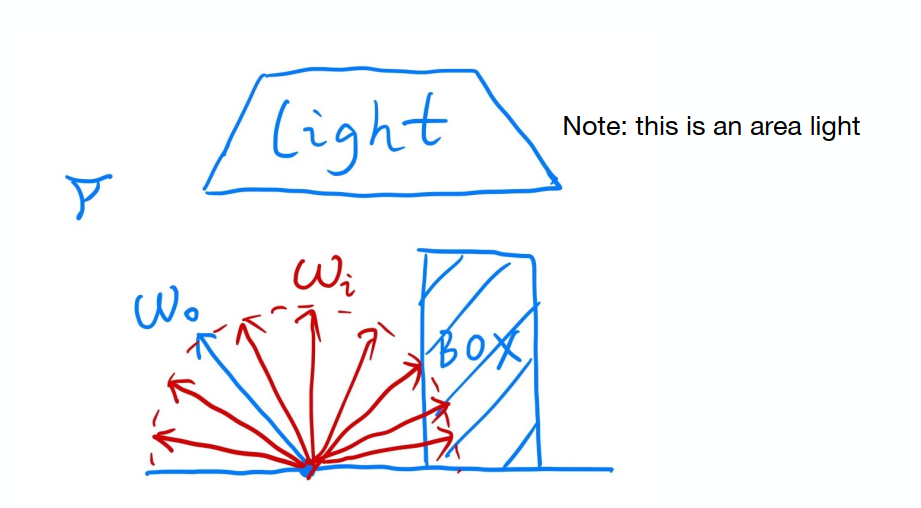
②接收到的光照来源于光源还是物体反射不做区分，所以要做递归

下面就来解决这些问题。

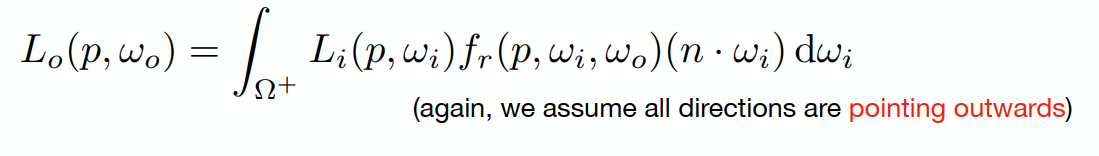
## 2、A Simple Monte Carlo Solution（Direct illumination直接光照）

在这一个简单场景中，我们就考虑清楚对于一个点（像素），他的直接光照是什么。

1. 有可能有其他物体Box会挡住光
2. 有一个相对较大的面光源area light
3. 观测方向:着色点到摄像机的方向ωo
4. 各个不同的进来的光(入射光方向）:ωi（与Blinn-Phong一样，认为方向都是从着色点出发往外打）

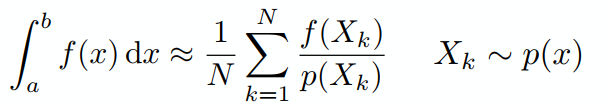


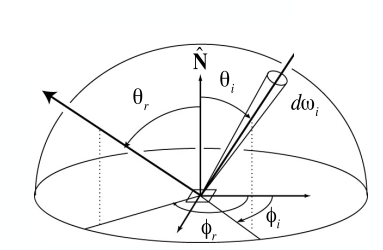
我们只考虑直接光照，不考虑其他物体反射过来的光，因此这里Li就是光源的光，且没有Le。直接光照渲染方程可简化如下(p为着色点，wo为观测方向立体角，wi为光线入射方向立体角，fr为BRDF)：



这个式子实质是在半球上的一个积分，那么在这里我们就可以用蒙特卡洛积分去解这个式子。

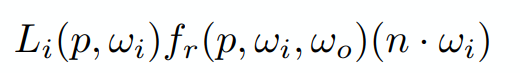
蒙特卡洛积分：





在这里我们假设对以上半球进行均匀的采样，那么与渲染方程近似的蒙特卡洛积分式各个部分如下：

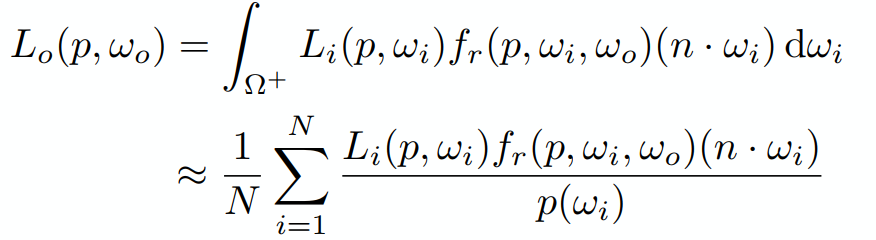
蒙特卡洛积分式的f(x)为：



蒙特卡洛积分式的p(x)(概率密度：假设为球面均匀采样)为：



那么直接光照的渲染方程可写为以下形式：



对照着上面的式子，可以把着色算法写出来：

shade(p, wo) //对于任意着色点p，考虑沿着ωo方向出来算Lo

Randomly choose N directions wi~pdf //在半球上选N个方向采样

Lo = 0.0 //先将最后的结果初始化

For each wi //对于任意一个选中的方向

Trace a ray r(p, wi) //考虑从p点向那个方向ωi连出一条光线

If ray r hit the light //如果这根光线打到了光源

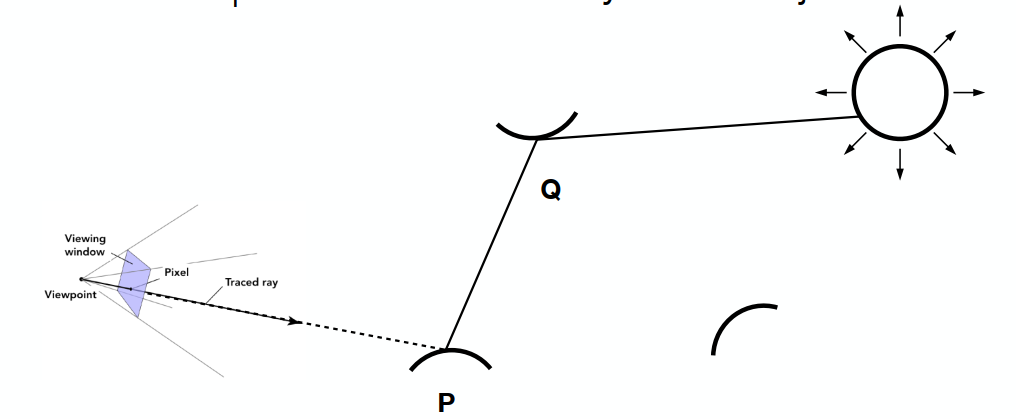
Lo += (1 / N) \* L\_i \* f\_r \* cosine / pdf(wi) //那么就把这个式子写出来（光线打到了光源自然就知道了各个值）

Return Lo

到这里基本的直接光照部分已经结束了

## 3、Introducing Global Illumination

刚刚解决的只是直接光照的问题，但是我们最后的结果要是全局光照，因此再引入间接光照。



间接光照要考虑光如果从Q点反射到P点，该怎么办？

由于渲染方程中不区分光源的Radiance和物体反射的光的Radiance，所以我们同样也可以把Q点看成点光源，Q点反射到P点的Radiance就是Q点光源发出的Radiance。这个时候我们类比，就好像是从P点观察Q点，算出Q点的直接光照即可。

那么就可以对刚刚的算法做一些修改：

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 //如果打到的是物体（不是光源）

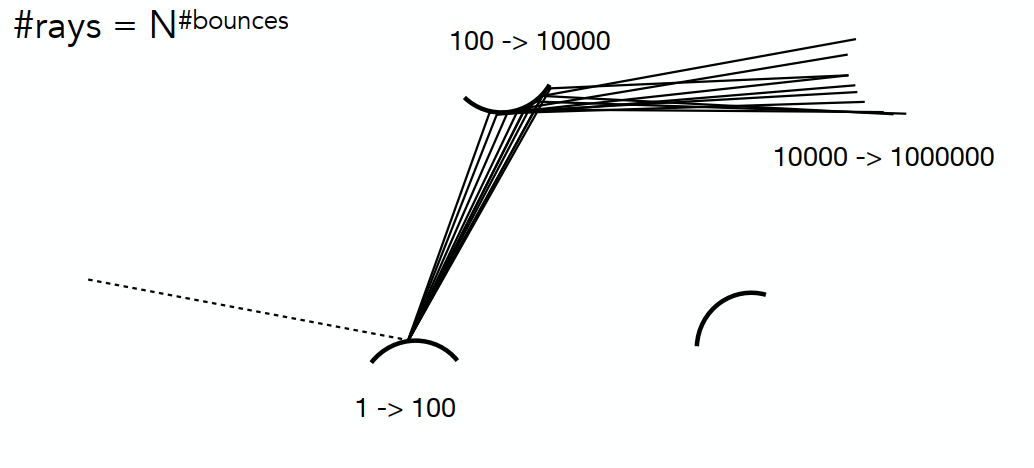
Lo += (1 / N) \* shade(q, -wi) \* f\_r \* cosine/ pdf(wi) //考虑在q点以-ωi方向进来的光写出方程，这里相当于把Q点直接光照的结果作为P点入射过来的光照

Return Lo

但是这样问题还没有结束！

#### Problem 1: Explosion of #rays as #bounces go up:

如果以这样的方式打出光线，光线的数量会爆炸。



P点往四面八方打出了N根光线，碰到物体后每条光线又会打出N根光线，碰到物体后每条光线又会打出N根光线，会造成光线数按指数级增长，造成指数爆炸

##### Solution：N=1

所以N取多少才能保证不出问题？答案只能是N=1。

修改算法如下

shade(p, wo)

Randomly choose ONE direction wi~pdf(w) //随机往一个方向采样ωi

Trace a ray r(p, wi) //往一个方向打出一条光线

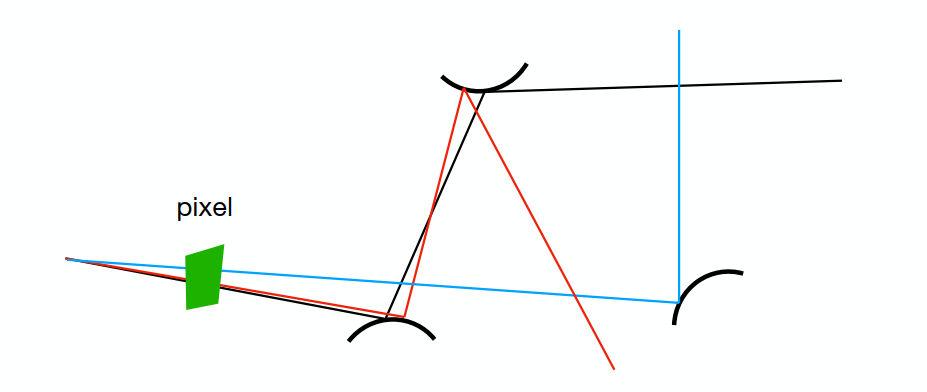
If ray r hit the light

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)

到此为止，用N=1来做蒙特卡洛积分，这就叫路径追踪。



虽然用这种方法的噪声会非常大，但是由于最后要的是一个像素最后的值是多少，穿过一个像素的路径会有很多条，这些所有的路径取平均即可。也就是说用足够多的路径（path）即可解决问题。（之所以叫path，是因为每次打到一个点上只会往一个方向去反射，不是一次产生一束，这就形成了一条连接视点和光源的路径。）

这个过程的操作步骤与光线追踪中的光线投射非常类似：

对屏幕中的每个像素值的算法可定义如下：

ray\_generation(camPos, pixel) //定义摄像机位置、要打出很多光线的像素

Uniformly choose N sample positions within the pixel //在这个像素内均匀取N个位置

pixel\_radiance = 0.0 //像素初始光照为0

For each sample in the pixel //对于任何一个选取的位置

Shoot a ray r(camPos, cam\_to\_sample) //从视点（摄像机）位置向样本位置连一条光线

If ray r hit the scene at p //如果光线打到了场景中某个位置

pixel\_radiance += 1 / N \* shade(p, sample\_to\_cam) //计算该点的着色

Return pixel\_radiance

其中shade函数的定义如下：

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)

但是这个算法是递归的，永远停不下来。

#### Problem 2: The recursive algorithm will never stop!

如果直接规定反弹多少次就停止，这样一种从中间直接砍下来的结果会导致能量损失，但是又无法模拟与真实的自然界一样的无限次反弹。因此人们引入了一种方法：

##### Solution: Russian Roulette (RR)（俄罗斯轮盘赌）

RR说明：一把左轮手枪可以装6发子弹，如果里面装了2枚子弹，那么生存的概率就是4/6。

通过用俄罗斯轮盘赌的方法，来决定一定的概率来停止继续往下进行追踪

某一个着色点最后的结果Radiance算出来的是Lo，我们希望中间可以停掉，但是算出来的结果不能出错，还是Lo。利用轮盘赌，首先自己定一个概率P (0 < P < 1)，以一定的概率P打出一条光线，最后得出的结果Lo再去除以概率P。那么另外的1-P的概率就不打出光线。这种方法的期望值仍然是Lo ： E = P \* (Lo / P) + (1 - P) \* 0 = Lo

将其以代码的形式表示出来:

shade(p, wo)

Manually specify a probability P\_RR //定义一个概率P\_RR

Randomly select ksi in a uniform dist. in [0, 1] //在计算机中随机取一个0到1的数ksi

If (ksi > P\_RR) return 0.0; //如果 ksi大于概率P\_RR，意味着这根光线不能生存

Randomly choose ONE direction wi~pdf(w)

Trace a ray r(p, wi)

If ray r hit the light

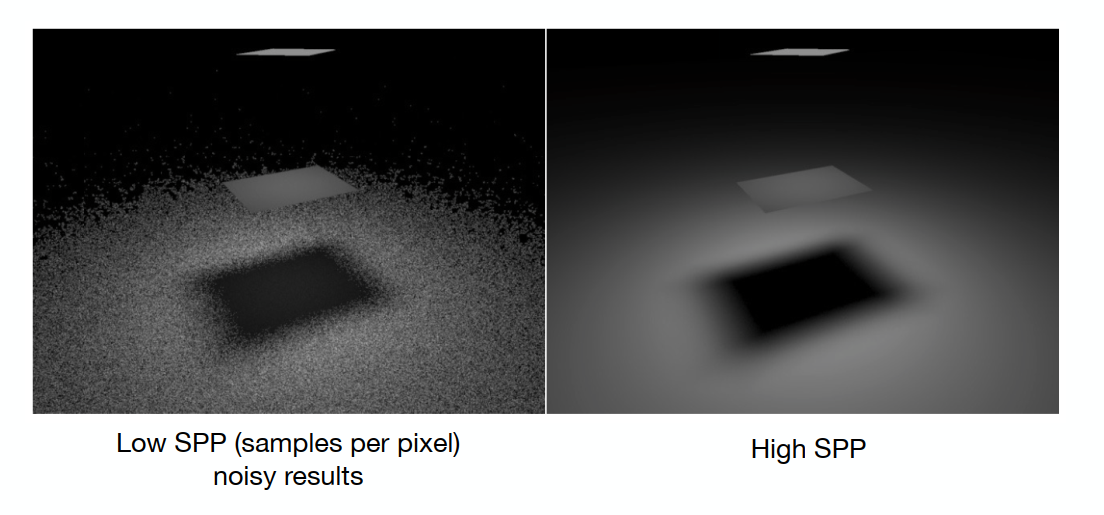
Return L\_i \* f\_r \* cosine / pdf(wi) / P\_RR //那么正常打光线的时候就去除以生存的概率P\_RR即可

Else If ray r hit an object at q

Return shade(q, -wi) \* f\_r \* cosine / pdf(wi) / P\_RR

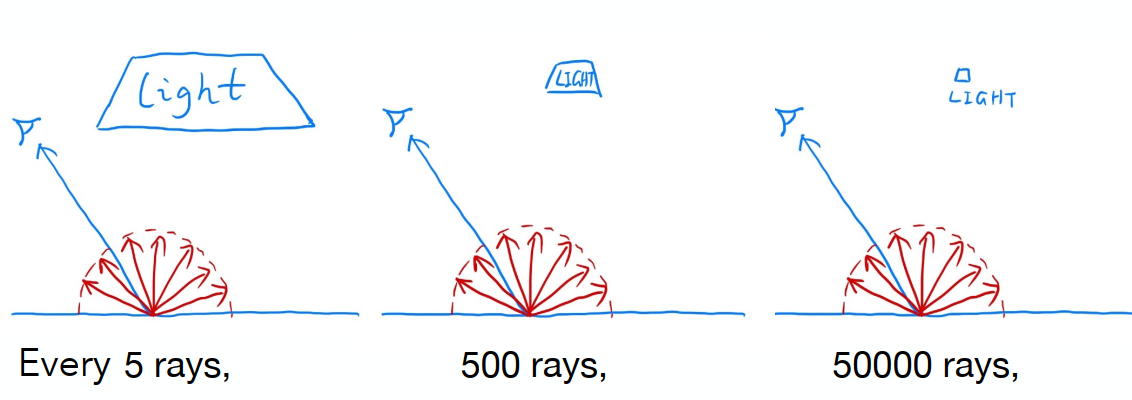
这样就会以一定的概率结束掉Else If的递归过程。到此为止，这已经是一个正确的路径追踪的方法了。但是还有一些问题，这并不是多么高效。

#### Problem3：Low efficiency



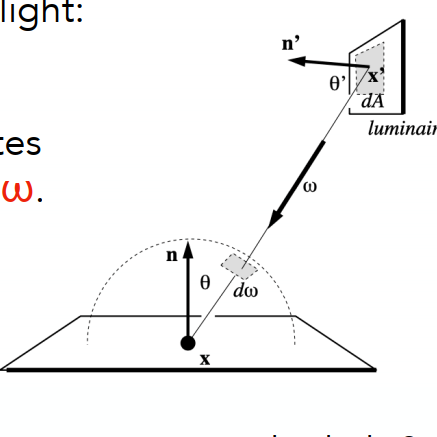
按照上述的方法，如果打出的光线少，渲染速度快，但是得到的质量会有很多噪声，如果打出光线多，得到结果好但是速度慢。我们希望Low SPP情况下质量依然能够很好。

##### Solution：Sampling the Light



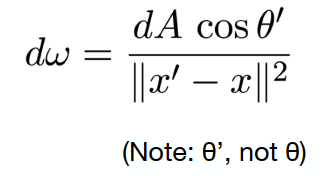
对于面积较大的面光源，可能我在着色点打出很少的光线就能碰到光源，但是如果面光源很小，那么我可能需要很多根光线才能有一根打到光源，这个就完全是看运气，就会造成很多光线浪费。

如果我们直接从光源上采样，那么所有光线都不会被浪费了

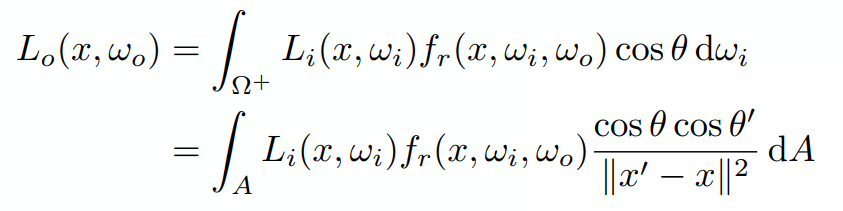


假设对于着色点在光源上采样，那么对于这个面光源来讲，光源面积是A，均匀采样的PDF就是1/A。但是渲染方程是定义在半球中立体角上的，而不是在光源上的。采样在光源上采样，积分在立体角上积分，如果要确保渲染方程还能用，就要把渲染方程写成在光源上的积分。所以我们只需要知道dω和dA的关系即可。

注意dA是光源上一个小的表面，dω是单位球上的立体角。我们之前说过立体角是半球上的面积除以半径的平方，也可以理解为把任何一个面积投影到单位球的表面上所对应的面积除以1单位长度的平方。那么把dA往单位球上作投影，即可算出：



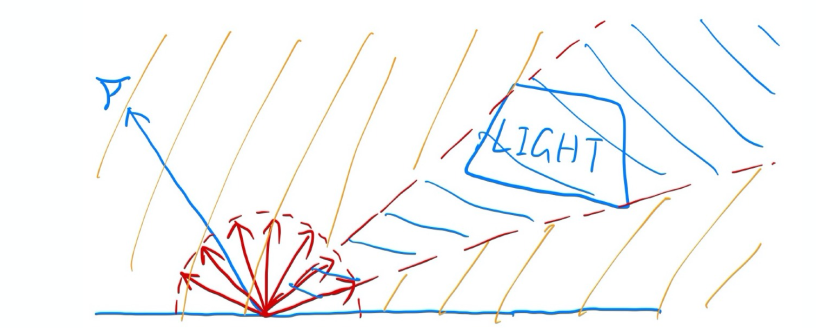
这样我们就可以把渲染方程成下述式子。



这时式子就变成了对光源进行采样，对光源进行积分。

之前我们是盲目的采样，打到打不到光源要看运气，现在我们可以直接对光源采样。

这时我们把着色结果分为两部分：



第一部分来源于对光源的贡献，对光源采样(这部分不涉及RR，因为这是光源直接作用于着色点；只有涉及到多次弹射才会有RR)

第二部分来源于其他所有非光源的贡献（如物体反射的光），这部分还用原来的方法做

这样就可以把算法重写。

shade(p, wo)

# Contribution from the light source. //光源对这一点的贡献(直接光照)

Uniformly sample the light at x’ (pdf\_light = 1 / A) //均匀地对光源进行采样

L\_dir = L\_i \* f\_r \* cos θ \* cos θ’ / |x’ - p|^2 / pdf\_light //在光源上积分

# Contribution from other reflectors. //其他所有非光源的贡献(间接光照)

L\_indir = 0.0

Test Russian Roulette with probability P\_RR //用俄罗斯轮盘赌测试

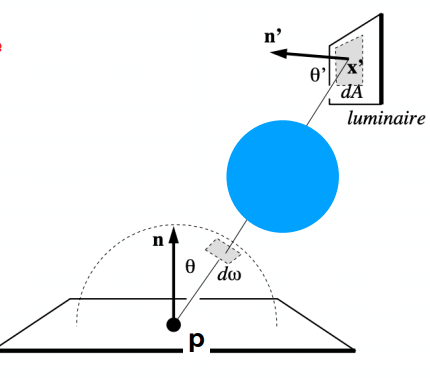
Uniformly sample the hemisphere toward wi (pdf\_hemi = 1 / 2pi) //如果通过了俄罗斯轮盘赌的测试，则发出一条光线

Trace a ray r(p, wi) //间接光照，从物体p向ωi方向打出一条光线

If ray r hit a non-emitting object at q //打到了点q，确定点q不是光源（因为光源的贡献已经算出来了）

L\_indir = shade(q, -wi) \* f\_r \* cos θ / pdf\_hemi / P\_RR

Return L\_dir + L\_indir//两部分加起来



如果光源和着色点之间被其他物体挡到，则不能算进去这个着色点在光源处的采样，我们在p点和光源取一条连线，判断中间有没有碰到其他物体即可。仅对算法稍加修改：

# Contribution from the light source.

L\_dir = 0.0

Uniformly sample the light at x’ (pdf\_light = 1 / A)

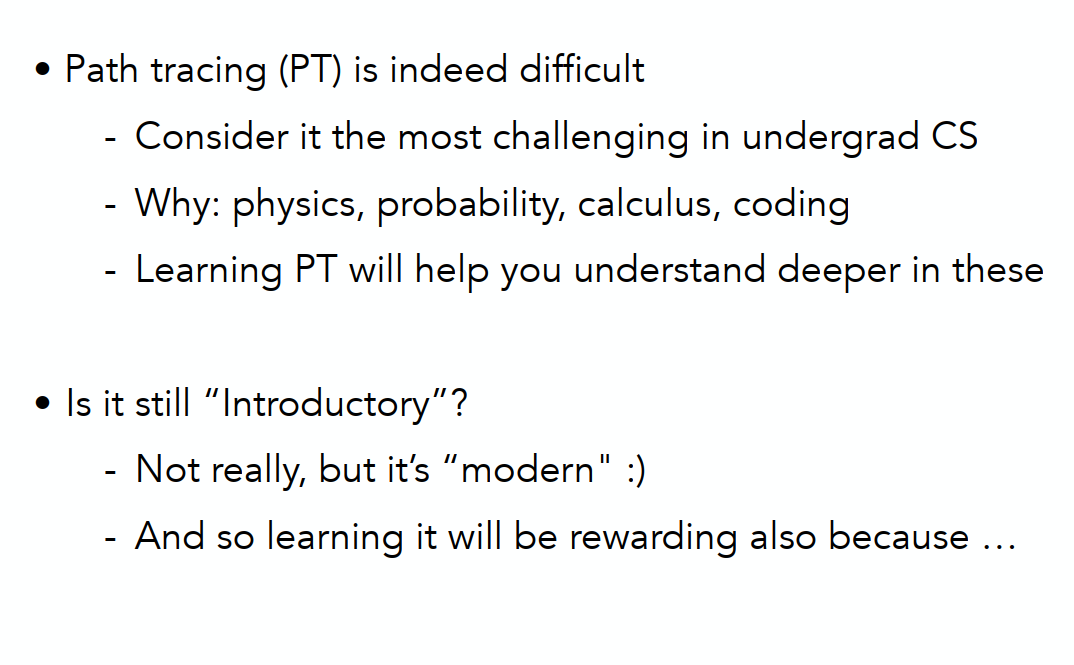
Shoot a ray from p to x’

If the ray is not blocked in the middle //光源与着色点直接没有遮挡

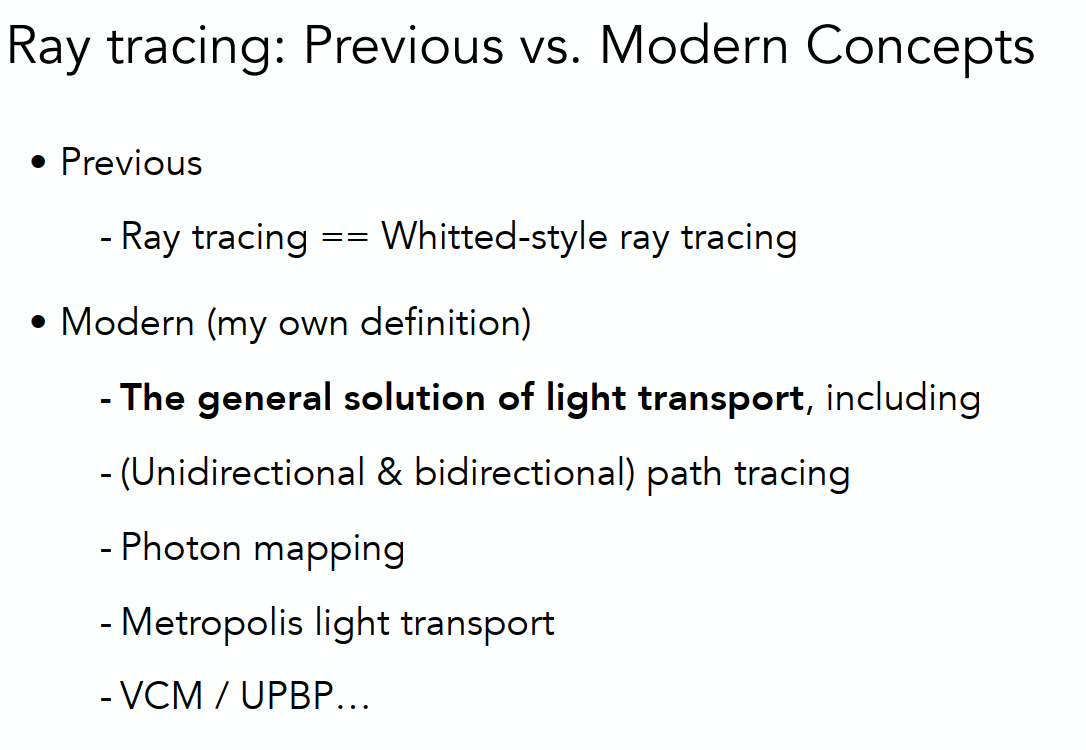
L\_dir = …

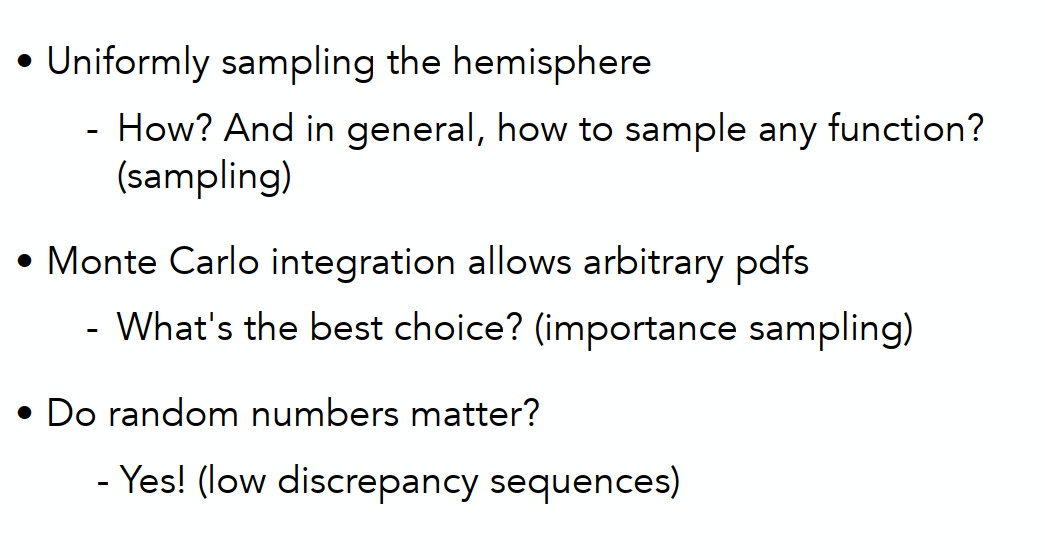
到此为止，路径追踪全部结束！

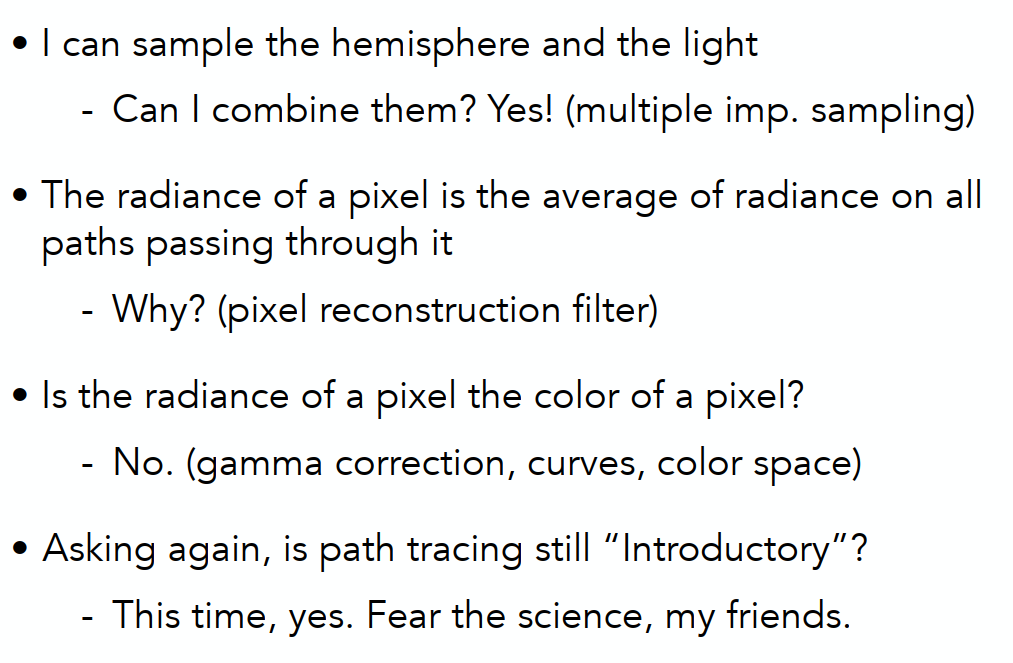
# 四、Some Side Notes



The Cornell box — http://www.graphics.cornell.edu/online/box/compare.html







可以看到，虽然路径追踪的大体思路讲完了，但是依然还有很多细节上的问题根本没有提到。整个路径追踪的内容太多了。现在的研究依然在向这个方向努力。