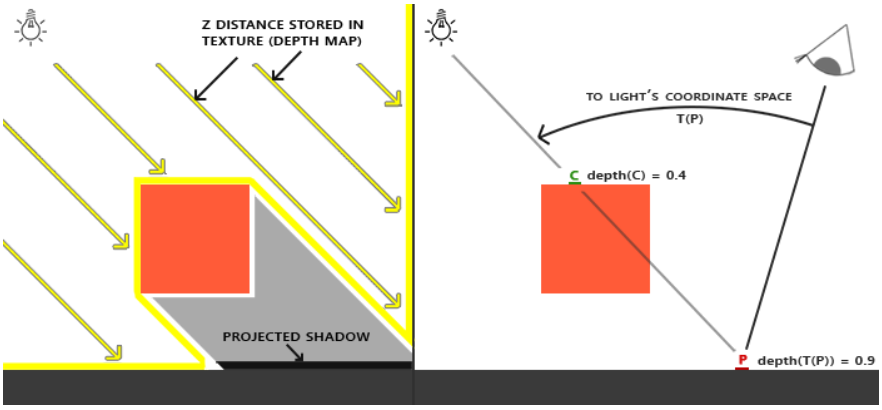
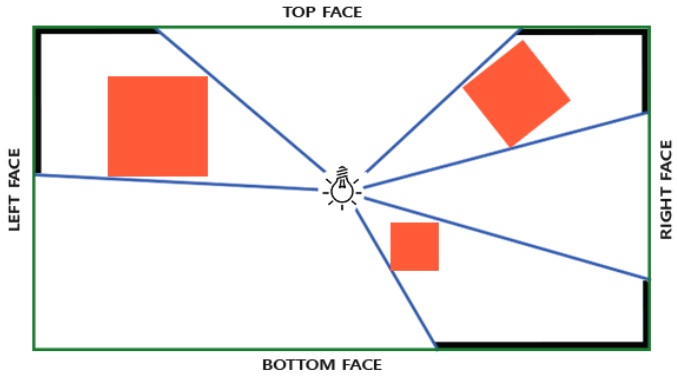
What's more

目前的着色内容的局限性

阴影与遮蔽

固定渲染管线总是在*fragment shader*中进行着色操作，这里的一个显著问题是每个*fragment shader*都只知道自己负责的片段的数据，而无法获得场景中物体的整体信息，带来的最直接的影响便是无法处理阴影和遮蔽问题，即无法直接判断一个片段的可见性，这会导致渲染的真实性大打折扣
为了解决这一个问题，一些渲染管线中会采用阴影映射的做法





其原理是先以光源为观察视角，渲染一张深度贴图，在开始着色时，用一个变换矩阵将片段位置变化至光源视角的坐标系中，将其深度(z值)与之前的阴影贴图中相比较，如果更高则说明这个片段在阴影中，从而不予着色
左图以平行光源为例展示了这一过程，更常见的点光源则需要将阴影贴图渲染至一个立方体贴图上（一个六面正方体的贴图，其优势是可以用一个3维空间矢量进行采样，是固定管线中用于抽象场景环境的各种信息的常用技巧），实际渲染时对该立方体贴图进行采样

另一个命题则是环境光遮蔽*AO*，其描述了物体表面被周围的几何体遮挡而变暗的现象，在实时管线的渲染中可以用一张环境遮蔽贴图*ao map*描述物体表面不同位置的被遮蔽程度，也有在屏幕空间进行采集遮蔽因子的兼顾效率和效果的技巧*SSAO*，有兴趣的同学可以直接查阅关于*SSAO*的相关资料

间接照明/反射/折射

*fragment shader*无法获知场景中物体分布的整体信息导致的另一大问题便是无法计算间接照明。回想一下我们之前的着色做法，都是由cpu传入光源数据，在*shader*中作为uniform值计算直接照明。但现实中光线会在物体中来回反射，真实的照明光源总是包括直接光源和间接光源，也许很多场景中给粗糙的物体着色时间接的照明并不对视觉效果的主要成分造成显著影响，但光滑的镜面反射和折射事件却需要体现映射周围环境的视觉效果。

考虑光滑的金属或镜面材质(回忆之前的多层材质模型，镜子其实也是一个折射的玻璃层下面是一个超高反射率的金属水银层……？)，现实中它们的表面会映照着周围的环境内容。计算非常简单，计算视线方向(出射光线)相对于法线的反射方向作为入射方向，逆向延伸至环境第一个接触的物体，将其颜色作为这一片段的颜色便可以体现“环境映射”的效果，折射也只需要用斯涅耳定律计算下折射方向即可。

然而对于一个完全镜面反射片段来说，如果只有直接光源的信息，那么如果视线方向不和入射方向正好关于法线对称，那么着色效果则是完全的黑暗，现实中金属的光泽感和经过玻璃后的扭曲透射效果就无从模拟了。

实时管线中的how to

实时渲染管线固然对性能有苛刻的限制但这并不代表其甘于只绘制受限的场景，现实中，尤其是游戏领域，创造了无数的*tricky*技巧来弥补着色过程本身的不足，让我们在60FPS的电子游戏世界也能发出“啊，这光，这水”的赞叹

Deferred Rendering

延迟渲染是次时代实时渲染工业圈的一个重要技术
传统渲染流程又称前向渲染forward rendering，一次draw call绘制所有片段（draw call指一次cpu向gpu发送数据，指定shader，渲染至屏幕的过程），而延迟渲染则分为两个pass
第一个pass创建一系列G缓冲，将位置、法线、贴图颜色等在**屏幕空间**的信息渲染至G缓冲的若干帧缓冲中（就是二维贴图啦），第二个pass直接在**屏**

幕空间组合这些信息，计算光照，获得渲染结果
具体介绍网上有非常的多的资料，*LearnOpenGL*里也有相应章节，有兴趣可以自行了解具体步骤，并多思考其利弊所在

SSAO

前文也提到，既然GPU运行过程中，每个片段不知道别的片段的信息，既然不能获得整个模型的全貌，我如何计算模型各个地方是否会被“遮蔽”呢？
在Deferred Rendering框架中，第二个pass里已有了屏幕空间的物体的完整信息，而且只需要对贴图采样即可，因此SSAO意为屏幕空间的环境光遮蔽，便利用了这一特性
对每个片段，在其法线对应的半球体进行采样，部分采样点会处于别的物体内（深度判断），部分在外，其比值就可以作为遮蔽系数，将物体内部的折角处渲染得更暗
具体内容有兴趣也可以自行搜索介绍，*LearnOpenGL*里也有相应章节

烘焙 && 光照探针

前者预渲染直接和间接光照的结果至贴图，后者预渲染至一个探针供其他物体在取用，游戏引擎，如Unity3D、UE4中相当常见，在这里不多赘述

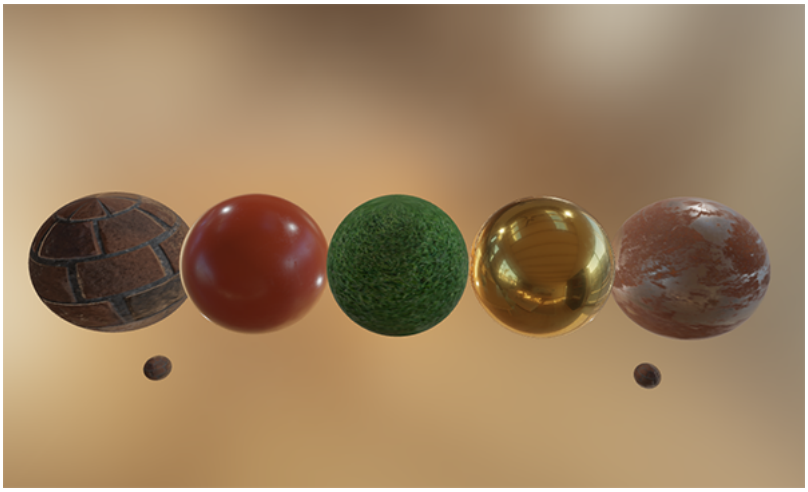
AO贴图、法线贴图、视察贴图……

其实也是一种预计算的方法，用贴图存储物体材质的各类细节供着色时使用

漫反射辐照与镜面IBL

IBL(Image Based Lighting)意为基于图片的光照，即把贴图（一般是作为环境的立方体贴图）本身当做光源模拟整个环境对片段的着色影响。如果物体表面并非完美镜面（完美镜面的入射方向唯一确定），依然需要用一次蒙特卡洛方法采样环境贴图，但这个采样的分布值也是可以预计算并写入纹理在片段着色时直接取用的

如果对在OpenGL中实现漫反射辐照和镜面IBL的效果有兴趣可以参考*LearnOpenGL*的Diffuse irradiance和Specular IBL章节



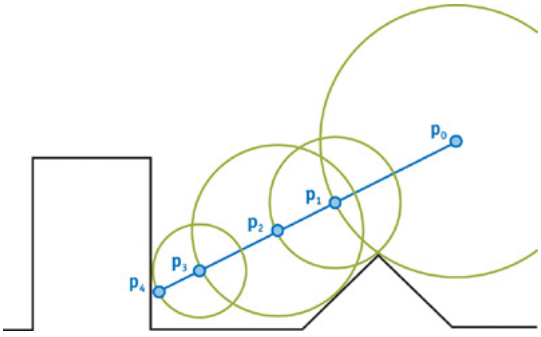
上图是一个糅杂了各种贴图和IBL的渲染结果

离线渲染

Ray tracing && Ray casting && Ray marching

光线追踪是一个很大的议题，这里没必要展开讲，在离线渲染中因为不需要追求实时性，因此可以利用光线可逆的原理从相机视点出发发射向场景中发射光线，迭代每一次在物体表面的反射直到接触光源本身，是一种暴力求解渲染方程的做法。

*Ray casting*和*Ray marching*是两种做光线求交的方法，前者用计算一个射线和一个空间内三角面的交点，一般需要依赖一些加速结构防止对空间内面片没必要的遍历，后者则是用逼近法求交点，但基本上用于方程描述的几何体构成的场景而不会在“三角形片世界”使用



光线追踪因为暴力求解了渲染方程，所以一定既有直接照明的效果又有间接照明的效果，某种程度上也是逼近物理真实情况的做法，随着现在*GPU*性能的提高离线渲染的效果会越来越逼真

Path tracing && 蒙特卡洛采样

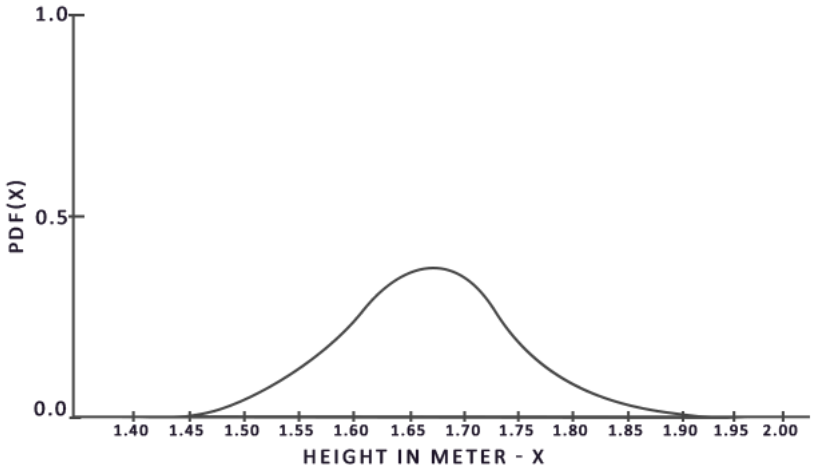
$$L_o(P,\omega_o) = L_e(P,\omega_o) + \int_\Omega f_r(P,\omega_i,\omega_o)L_i(P,\omega_i)cos(\theta_i)d\omega_i$$

$\int_\Omega$符号要求我们对法线所在的半球面进行积分，但是在离散的计算机世界求连续的积分只能用采样求和的方法，即将其转化为$\sum_\Omega$，采样得越多，离散求和的结果越向积分式的真实解收敛，但考虑到性能(计算时间和内存限制)问题，迭代渲染方程时只能用一定限度的采样数，因此实际计算中，我们必须用某些方法加速收敛过程

假设我们对半球面均匀地采样(其实这是第一个问题，如果有一个2维均匀分布的随机变量，如何将其均匀映射到一个半球面上……？)，但实际上贡献照明的角度可能只有极少的角度，从而在采样数受限时导致严重的噪声(*noise/aliasing*)，而蒙特卡洛法则可以解决这一问题

$$O = \int_a^b f(x)dx = \frac{1}{N}\sum_{i=0}^{N-1}\frac{f(x)}{pdf(x)}$$

并不是什么玄乎的东西，很直观的道理——光照成分贡献多的角度多放一些采样点，少的角度少放一些采样点就好啦。因为采样点的分布不是均等概率了，因此为了积分计算的正确性在求和时需要除以采样点出现在此处的概率$pdf(x)$



这个思想很直观，也很容易实现，但我们需要考虑一个问题，即假设我们已知概率密度函数$pdf(x)$，给定一个独立随机变量U（U的概率密度函数处处为1），那么如何对其做一个映射$U \to X$，使映射后的分布符合给定的$pdf(x)$

直接上结论：$X = CDF^{-1}(U)$

证明如下：首先假设$Y = CDF^{-1}(U)$，有$CDF(x) = Pr(X < x)$

$$Pr(Y < x) = Pr(CDF^{-1}(U) < x) = Pr\{U < CDF(x)\} = CDF_u(CDF(x)) = CDF(x)$$

因此$X = Y$

在图形学领域，经常需要在三维空间里求解渲染方程，那么其材质的概率密度函数往往是二元的，即方位角ϕ和天顶角θ，并且ϕ和θ之间不是互相独立的对于非独立的二元变量，已知$pdf(x,y)$，需要使用一对二元的独立随机变量ξ_1和ξ_2分别映射：

$$pdf(x) = \int pdf(x,y)dy$$
$$pdf(y|x) = \frac{pdf(x,y)}{pdf(x)}$$

在此基础上我们就可以根据已知的材质特征对其进行蒙特卡洛“有偏”采样，有兴趣的同学可以看下我附录的隔壁计科《图形绘制技术》课程的PPT，里面详细解释了数学推导过程，和一些典型分布的有偏映射的计算过程

Cook-Torrance微表面模型的蒙特卡洛有偏采样

BRDF已经告诉了我们在已知入射光线时，出射光线的辐射率强度在半球面上的分布(回忆实验2的一些示例图)，自然，我们希望数值高的角度多放采样点，低的少放采样点，为0的地方直接不予采样，最完美的方法便是采样点分布的概率*pdf*和BRDF的分布完全一致

考虑渲染方程对入射光线的积分式的系数$f_r(\omega_i,\omega_o)dot(\omega_i,n)$，一般来说因为几何函数、菲涅尔系数等项，微表面模型的方程无法直接求得其积分的解析解，所以我们先采样一个微表面的法线方向，即，ω_i和ω_o的中间向量ω_m，再用反射公式和已知的ω_o，求得ω_i，作为有偏采样的一个样本（出自之前介绍过的经典论文*Microfacet Models for Refraction through Rough Surfaces*)

回忆在作业3中，对Cook-Torrance模型最终的BRDF的推导式的某个中间过程（出自《PBRT》）：

$$L(\omega_o) = \frac{F(\omega_o)L_i(\omega_i)d\omega_i dA}{d\omega_o dAcos\theta_o}D(\omega_m)cos\theta_m d\omega_m$$

因此对ω_m的采样的概率密度函数定义为$pdf(\omega_m) = D(\omega_m)cos\theta_m$，可以获得非常好的有偏采样结果（根据法线分布函数的定义，这个pdf天然在半球面内积分为1，只要其积分有解析解即可，之前介绍的的beckmann和ggx都有积分式的解析解）

又有：

$$pdf(\omega_i) = pdf(\omega_m)\left\|\frac{\omega_m}{\omega_i}\right\| = \frac{D(\omega_m)dot(\omega_m,n)}{4dot(\omega_o,\omega_m)}$$

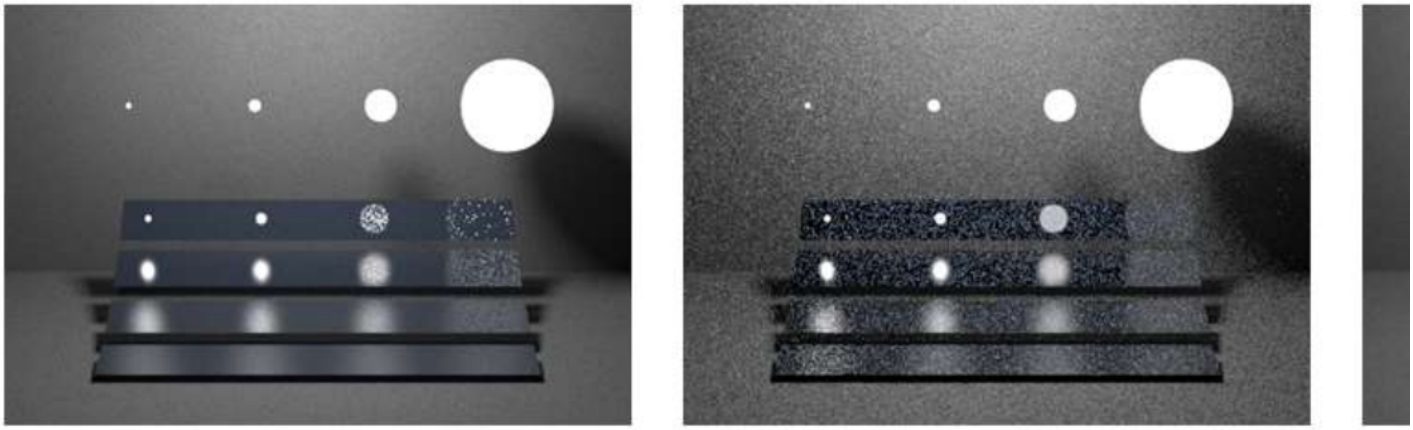
因此最后渲染方程可以被拆解为：

$$\frac{1}{N}\sum^{N}\frac{L_i * F * G * dot(\omega_o, \omega_m)}{dot(\omega_m, n) * dot(\omega_o, n)}$$

最终，我们可以用蒙特卡洛的有偏采样的方法，快速收敛渲染方程的数值解，从而渲染全局光照

论文*Microfacet Models for Refraction through Rough Surfaces*中也直接给出了beckmann和ggx的法线分布函数，从两个独立随机变量ξ_1和ξ_2映射到θ和ϕ的公式

Path tracing中关于Mutiple Importance Sample的问题



我们对所需要渲染的物体表面上某一点的法线所在的半球面进行积分，那么无论采样什么方向，都有两种可能：

1. 方向指向光源（这里的光源一般都不再是点光源，而是一个自发光体）
2. 方向指向其他非光源物体，接收其在这个方向反射出来的光线

首先，为了能尽快渲染出场景，采样数有限的情况下，直接采样可能会产生大量的噪点（中间图）

考虑直接光源的影响远大于间接光源，因此可以对直接光源和间接光源分别采样，对直接光源来说，直接在发光体上均匀选点作为入射方向，间接光源则使用之前一直在讨论的蒙特卡洛有偏采样，但如果正好采样到了直接光源的方向，需要进行丢弃，否则这个方向的光线会被多计算一次（左图为结果）
这里的问题在于，直接光源上的采样点也有限，而且其分布是纯随机的，没有像蒙特卡洛有偏采样那样，集中在法线分布函数概率高的方向，万一采样的点方向比较偏，计算出来的结果就会小，反之可能会大，结果就是可能出现明显噪声（左图的右上角）

Mutiple Importance Sample其思想为在计算直接光照时，乘上系数w_light：

$$w_{\text{Light}}(p_{\text{Light}}, p_{\text{BRDF}}) = \frac{p_{\text{Light}}}{p_{\text{Light}} + p_{\text{BRDF}}}$$

同理，在计算间接光照时，若击中光源，则乘上系数w_{BRDF}：

$$w_{\text{BRDF}}(p_{\text{Light}}, p_{\text{BRDF}}) = \frac{p_{\text{BRDF}}}{p_{\text{Light}} + p_{\text{BRDF}}}$$

结果为右图，同样的采样数，效果最好

Nori

关于基于*Path tracing*的离线渲染实现在这里推荐一个很好的练习内容——由《PBRT》的作者执教的《Advanced Computer Graphics》的课程作业Nori，其中内容基本上是hand-by-hand的教学，如果全部完成能对整个渲染过程有清晰的理解
上一part给的三个渲染图均出自我完成的此项目渲染出的结果

UE4中在实时管线里使用预计算贴图的方式高效率实现基于微表面的IBL的方法

实时管线里因为效率原因无法直接在片段着色器生成采样点，因此UE4介绍了一种通过预计算LUT贴图的方法，高性能实时地从环境贴图中，生成全局光照的着色效果

建议直接看这篇论文Real Shading in Unreal Engine 4的前半部分，这篇文档也只是将其中的内容搬过来

最关键的地方是利用了这个近似公式：

$$\frac{1}{N}\sum^{N}\frac{brdf * L_i * cos\theta_i}{pdf} \approx (\frac{1}{N}\sum^{N}L_i) * (\frac{1}{N})\sum^{N}\frac{brdf * cos\theta_i}{pdf}$$

左半边部分对立方体贴图的每个方向为原始法线方向，做一次基于法线分布函数的蒙特卡洛有偏采样即可
其中有参数粗糙度roughness，可以用的方法是对几个层级的roughness分别映射到不同的mipmap级别上

右半边部分要做一个拆解：

$$F_0 \int \frac{brdf}{Fresnel}(1-(1-\omega_o \cdot \omega_m)^5)cos\theta_i d\omega_i + \int \frac{brdf}{Fresnel}(1-\omega_o \cdot \omega_m)^5 cos\theta_i d\omega_i$$

再把$\omega_o \cdot \omega_m$近似为$\omega_o \cdot n$，积分式就剩下粗糙度roughness和$\omega_o \cdot n$两个参数，可以在一张二维LUT表中记录

LearnOpenGL的镜面IBL章节部分给出了这个方法的代码实现，我基于作业框架拓展的ShadingSandbox也实现了这个方法