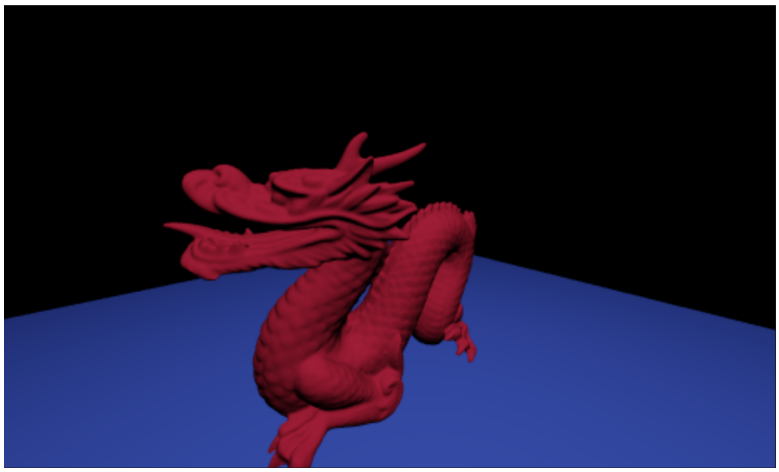
PBRT 系列 8-反射与材质的初步了解

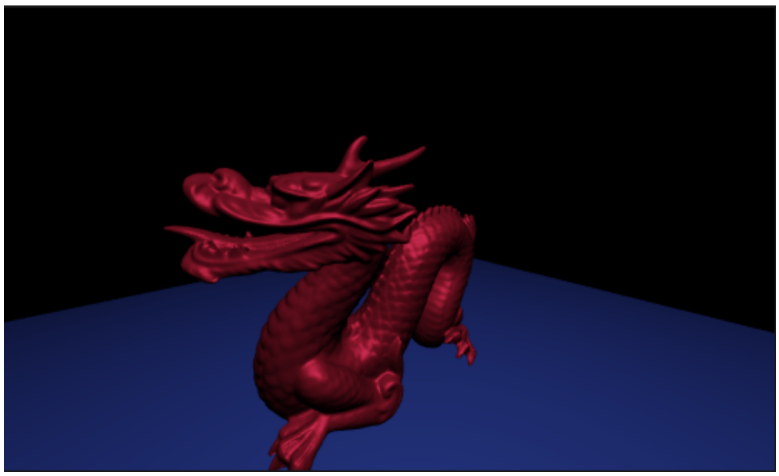
Dezeming Family

2021 年 3 月 4 日

因为本书是电子书，所以会不断进行更新和再版（更新频率会很高）。如果您从其他地方得到了这本书，可以从官方网站：https://dezeming.top/ 下载新的版本（免费下载）。

本书目标：学习 PBRT 的反射模型以及最基本的朗伯反射。学习材料类的基本接口。学习纹理类基本接口，实现最简单的常量纹理类。实现整个包含了材质和纹理的渲染流程，并渲染出下面的图像：





本文于 2022 年 7 月 9 日进行再版，提供了源码。源码见网址 [https://github.com/feimos32/PBRT3-DezemingFamily]。

前言

材质是 PBRT 中最重要的部分，PBR 本意为基于物理的渲染，物理材质是 PBR 中重要的一环，所以本书介绍的内容比较重要，这里包含了材质的基本定义和接口。本书我会尽可能多地描述材质的基本结构和属性，能移植的我们就一边讲解知识一边移植，否则最后再一起移植的话恐怕会吃不消。

前面的系列书中，PBRT 的内存管理、误差界定以及采样器的内容可以省略着看，知道基本流程即可，但是材质这部分内容要仔细弄懂 BxDF 类的每个细节的（至少，本书除了 BSDF 里面的几个我提到的可以暂时忽略的函数，其他所有的描述都要看懂）。

这本书我们会把最基本的材质接口和反射接口实现，并实现一个最基本的纹理——常量纹理，然后我们就用目前构建好的全部体系，渲染出色彩更丰富、稍微好看一点的图像。

本书的售价是 5 元（电子版），但是并不直接收取费用。如果您免费得到了这本书的电子版，在学习和实现时觉得有用，可以往我们的支付宝账户（17853140351，可备注：PBRT）进行支持，您的赞助将是我们 Dezeming Family 继续创作各种图形学、机器学习、以及数学原理小册子的动力！

目录

一　辐射度量学

本章简单回顾一下辐射度量学方面的内容。

1 1　辐射度

关于辐射度量学的资料有很多，一搜一大堆，而且表述基本上都是一致的（这里有必要说一句，这里的表述我是指英文表述，比如 irradiance 和 radiance，中文翻译的倒是乱七八糟的）。

在 PBRT3[1] 书的颜色与辐射度量学一章中第 4 节描述了能量、立体角与辐射度之间的关系。第 6 节描述了表面反射、BRDF 的简单介绍。这些介绍的内容希望大家都能够学会（但不必是现在）。

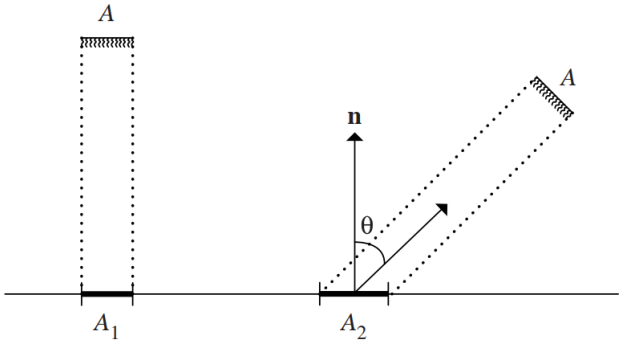
我强调几个地方：irradiance（记为 E）表示的是单位面积的通过的能量功率，也就是单位面积单位时间通过的光能:

$$E(p,n)=\frac{\Phi}{A} \tag{一.1}$$

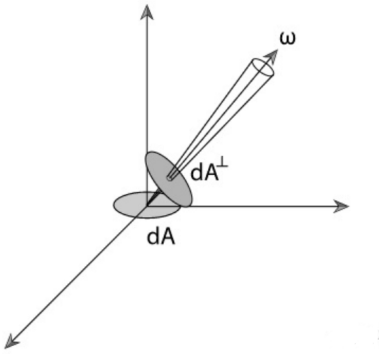
radiance（记为 L）表示的是单位面积单位立体角通过的能量功率，即单位面积单位时间单位立体角垂直于平面通过的光能（单位垂直立体角）：

$$L_i(p,n)=\frac{\Phi}{d\omega dA^{\perp}}=\frac{\Phi}{cos(\theta)d\omega dA} \tag{一.2}$$

因为光可能斜着照射，所以需要乘以 cos 值。我们以下面为例：



不同方向照射到某单位面积的光不同。假如同样是功率强度为 T 的光，斜着照射到表面时，照射的面积就更大，单位面积接受到的功率就更小。所以当单位立体角下的表面功率相同时，立体角越偏，则说明该方向入射光的功率越大：



$$E(p,n)=\int_{\Omega}L_i(p,\omega_i)|cos\theta_i|d\omega_i \tag{一.3}$$

BRDF 表示为：

$$f_r(p,\omega_o,\omega_i)=\frac{dL_o(p,\omega_o)}{dE(p,\omega_i)}=\frac{dL_o(p,\omega_i)}{L_i(p,\omega_i)cos\theta_i d\omega_i} \tag{一.4}$$

其中，$cos\theta_i = (n \cdot \omega_i)$。

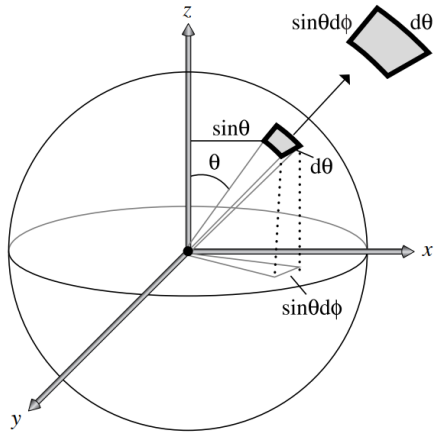
需要注意的是 BRDF 并不是辐射亮度 (radiance) 与辐射亮度的比值，而是辐射亮度与 irradiance 之间的比值，这是由渲染方程决定的。因为需要对各个不同方向立体角传来的 radiance 进行积分：

$$L_o = \int_\Omega f_r(p, \omega_o, \omega_i) dE(p, \omega_i) = \int_\Omega f_r(p, \omega_o, \omega_i) L_i(p, \omega_i) cos\theta_i d\omega_i \tag{一.5}$$

这样，两边求导，就能得到 BRDF 的表示形式了。

1 2　球面积分

使用球坐标系求球的面积时，我们以下图为参考。转换为 xyz 坐标时，$x = rsin\theta cos\phi$，$y = rsin\theta sin\phi$ 和 $z = rcos\theta$。求面积时，面积微元等于 $rsin\theta d\phi$ 乘以 $rd\theta$。



因此我们可以计算得到：

$$\int_0^{2\pi} \int_0^{\pi} r^2 sin\theta d\theta d\phi = 4\pi r^2 \tag{一.6}$$

当 r=1 时，我们得到单位球体的表面积等于 4π。

我们把对半球上的积分记为：

$$\int_\Omega f(\omega) d\omega \tag{一.7}$$

对于后面我们要做的 BRDF 半球积分中，并不是直接对 BRDF 进行半球积分，而是：

$$\int_\Omega f(\omega) cos(\theta) d\omega \tag{一.8}$$

这里是要附带上 radiance 表示的 $cos\theta$ 值。

二　初识反射模型

表示表面反射特性的 BRDF 和表示穿透特性的 BTDF 统一用 BSDF 表示，不同的 BRDF 和 BTDF 构成不同的材质。反射与透射可能会受到表面特性的影响，比如表面纹理和凹凸映射，我们暂时不予考虑。BSSRDF 次表面散射我们暂时也不考虑。

为了目录更清晰，我们把 reflection.h 和 reflection.cpp 文件的内容移植到 Material 目录下（见我们的源码）。

2 1　反射模型概述

我们的反射模型有多种来源：

第一种是真实测量的数据，就是用一束光打到材质上，然后从各个角度记录反射后到达传感器的亮度，多用于实验室的数据。

第二种是现象型模型：试图描述真实世界表面的方程可以非常有效地模仿它们，这些类型的 BSDF 特别易于使用，因为它们往往具有修改其行为的直观参数（例如，“粗糙度 roughness”）。

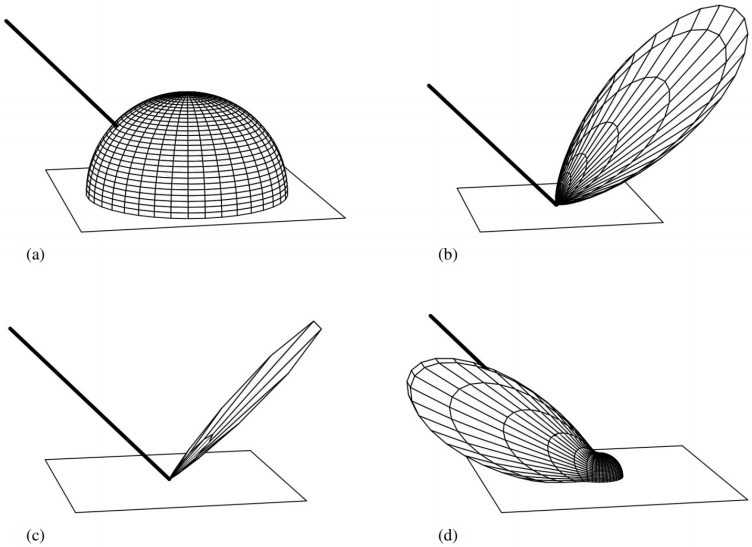
第三种是模拟：有时表面成分的基础信息是已知的，例如，我们可能知道涂料是由悬浮在介质中的某种平均大小的有色颗粒组成，或者一种特定的织物是由两种类型的线组成，每种线都具有已知的反射特性。在这些情况下，可以模拟来自微几何体的光散射来生成反射数据。这种模拟可以在渲染过程中进行，也可以作为预处理进行。在预处理之后，它可以适合于渲染过程中使用的一组基函数。模拟方法涉及了多种学科，例如计算化学和生物学、材料学等，我们暂时不考虑模拟有关的内容。

还有物理（波动）光学：利用光的详细模型导出了一些反射模型，将其视为波，并计算麦克斯韦方程组的解，以找出它如何从具有已知特性的表面散射（闫令琪博士的实验室有不少这方面的研究和渲染框架，大家有兴趣可以参考一下，对于传统工科的学生可能很难，但对物理系学生来说其实并不难）。然而，这些模型的计算成本往往很高，而且通常不会比基于几何光学的渲染应用模型更精确。

几何光学：与模拟方法一样，如果已知表面的低层次的散射和几何特性，则有时可以直接从这些描述中导出闭式反射模型。几何光学使得模拟光与表面的相互作用更容易处理，因为像偏振这样复杂的波效应可以忽略（其实在一些渲染，比如发光的显示屏和常见的金属材质中，偏振也会给最终渲染结果带来较大的改变）。

曲面着色器作为材质子类的一种方法实现，并负责确定曲面上特定点处的 BSDF；它返回一个 BSDF 对象，该对象包含它已分配并初始化的 BRDF 和 BTDF，以表示该点处的散射。

表面反射可以分为四大类：漫反射（diffuse）、有光泽的镜面反射（glossy specular）、完美镜面反射（perfect specular）和回复反射（retro-reflective）。大部分真实材质都是这四种反射的混合。

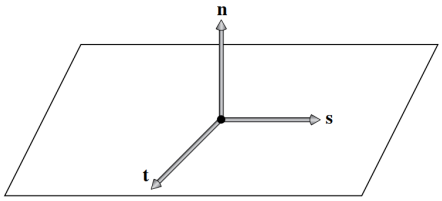


Glossy specular 表面比如较光滑的塑料材质。而 perfect specular 反射面将入射光散射到单个出射方向，例如镜子和玻璃是完美镜面反射的例子。最后，像天鹅绒这样的 retro-reflective 表面主要沿入射方向散射光。

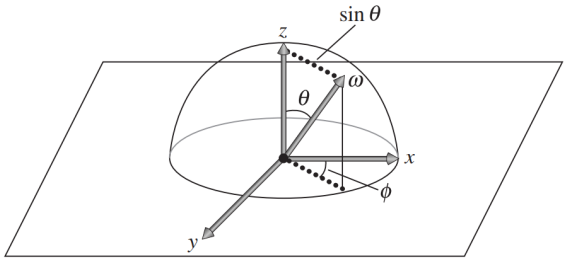
给定特定类型的反射，反射分布函数可以是各向同性的或各向异性的。大多数材质是各向同性的：给定一个点，沿着它的法向量旋转，反射光的分布不会变（换言之，反射方向的光亮度与“入射方向与法向量的夹角以及反射方向与入射方向的相对位置关系”有关，而与入射方向的具体方向无关）；而各向异性材料则会改变，比如光盘这种材料。我们以后遇到再进行详细解释。

2 2 反射的几何计算

我们之前在《形状与加速器》中介绍过 PBRT 中的变换，除了世界坐标系之外，计算光照中还有局部坐标系。其中法向量位于局部坐标系的 z 轴，两个切线方向分别是局部坐标系的 x 轴和 y 轴：



之后，BSDF 和 BRDF 都是在这个坐标系统上计算的，如果计算后产生了 Ray 的新方向会再变回到世界空间里。我们称这个局部坐标系为着色坐标系。着色坐标系也可以表示为球坐标形式 (θ, ϕ)，其中 θ 表示方向与 z 轴的夹角，因此在着色空间里，$cos\theta = (n \cdot \omega) = ((0,0,1) \cdot \omega) = \omega_z$。像 CosTheta 函数，Cos2Theta 函数和 AbsCosTheta 函数都是这么计算出来的，然后可以通过 cos 值计算出 sin 值和 tan 值。



同理，ϕ 的正弦和余弦值也都可以计算出来。

在着色空间里任意两个向量之间的 ϕ 夹角，则只需让 z 值都等于 0，然后根据点乘的方法求即可，参见函数 CosDPhi。

在研究 PBRT 坐标系之前，下面的前提我们必须要记住的，在这里我们标注为“着色几何 n”，现在大家可能对下面的内容没有什么直接感受，所以后面在源码中遇到时我会进行强调。

着色几何 1：入射光方向 ω_i 和视觉方向 ω_o 在转化到着色坐标系后都是要被单位化的，并且都是朝外的。

着色几何 2：根据 pbrt 中的惯例，表面法线 n 始终指向对象的“外部”，这使得很容易确定光是进入还是离开透射对象：如果入射光 ω_i 与 n 在相同的半球，则光就是射入的，否则就是射出的，因为 PBRT 不会把法向量反转到与 ω_o 相同的方向。

着色几何 3：BRDF 和 BTDF 不管它们的 ω_i 和 ω_o 是否在同一个半球。例如，对于 BRDF 来说，如果入射方向在表面上半球和出射方向在表面下半球，则不能进行反射。但在这里我们期望反射函数使用反射模型的适当公式计算并返回反射的光，忽略它们不在同一半球的细节。

着色几何 4：求交计算后得到的着色坐标系统可能会被改变，例如后面纹理中讲的凹凸映射。

2 3 BxDF 接口

我们将首先为单个 BRDF 和 BTDF 函数定义接口。BRDF 和 BTDF 共享一个公共基类 BxDF。因为两者具有完全相同的接口，共享相同的基类可以减少重复的代码，并允许系统的某些部分在不区分 BRDF 和 BTDF 的情况下通用地使用 BxDFs。

BxDF 类定义在 reflection.cpp 和 reflection.h 文件中。注意该文件同时定义了 BSDF 类，是一个 BxDF 对象的合集，表示一个表面会有多种反射模型构成。BxDF 类型一共有 5 种，定义在 BxDFType 枚举对象中，其中还有表示全部种类集合的 BSDF_ALL。BxDF 类的 MatchesFlags 方法判断该 BxDF 对象是否是某种类型。

BxDF::f(wo,wi) 返回光分布函数，这个接口隐含地假设不同波长的光是解耦的，一个波长的能量不会反射为不同的波长。通过这个假设，反射函数的影响可以直接用 Spectrum 来表示。（荧光材料则对这种假设不成立，需要这种方法返回一个 $n \times n$ 矩阵，该矩阵对光谱样本之间的能量传递进行编码（其中 n 是光谱表示中的样本数），但我们暂时不考虑荧光）。

对于完美镜面物体，反射光的分布需要特殊处理。BxDF::Sample_f() 不但处理完美镜面的分布，还负责在 BxDF 延多个方向反射光时进行采样。但是对于完美镜面来说，入射光方向 wi 并不是作为用来计算的参数，而是作为 Sample_f() 生成的值，因为对光采样得到的 wi 值几乎不可能正好是根据出射方向 wo 计算得到的完美镜面入射方向的值。同理，对于完美镜面来说，采样分布的概率密度函数 Pdf 也没什么用了：

```
1    virtual Spectrum Sample_f(const Vector3f &wo, Vector3f *wi,
2                              const Point2f &sample, Float *pdf,
3                              BxDFType *sampledType = nullptr) const;
```

反射比 ρ_{hd} 表示的是反射到某个方向的能量占所有入射的能量的比值:

$$\rho_{hd} = \int_{\Omega} f_r(p, \omega_o, \omega_i) |cos\theta_i| d\omega_i \tag{二.1}$$

该值经常使用蒙特卡洛方法求解。反射比在 BRDF 里是没有用到的，而是在 BSSRDF 次表面散射里会使用到，BxDF::rho 的参数就是一堆随机采样点。因为表面反射用不到反射比，所以我们暂时不考虑它。

半球-半球反射比则是反射到所有方向的积分：

$$\rho_{hd} = \frac{1}{\pi} \int_{\Omega} \int_{\Omega} f_r(p, \omega_o, \omega_i) |cos\theta_o cos\theta_i| d\omega_o d\omega_i \tag{二.2}$$

对于一个给定的 BxDF，有时候会根据 Spectrum 值来缩放它的光分布，ScaledBxDF 类就是这样，该类的两个成员变量：BxDF *bxdf 和 Spectrum scale 就是为了该功能，ScaledBxDF::f() 函数返回的就是 scale * bxdf->f() 值。

三　材质基础

PBRT 中每个基元都会绑定表面着色器，为表面上的点创建 BSDF 对象，这些计算在材料类 Material 里进行。

3 1　法向量与着色空间

BSDF 类在 reflection.h 和 reflection.cpp 文件里定义，传入参数中的两个法向量是 ns 和 ng，ng 就是交点处几何的法向量，ns 是计算着色时用的法向量。

```
BSDF(const SurfaceInteraction &si, Float eta = 1)
    : eta(eta), //与透明材质有关
      ns(si.shading.n),
      ng(si.n),
      ss(Normalize(si.shading.dpdu)),
      ts(Cross(ns, ss)) {}
```

在三角形求交中，ng 表示的几何法向量（g 是 Geometry 的意思）有两种情况：(1) 模型顶点不提供法向量，则法向量是根据三角形所在平面求垂直向量计算出来的；(2) 模型顶点提供法向量，那么它就是模型顶点的法向量。这个法向量是最朴素最原始的法向量。ns 是着色法向量，也有两种情况：(1) 模型顶点不提供法向量，则它跟 ng 一样；(2) 模型顶点提供法向量，则它会通过光线微分技术以及顶点的法向量来计算这个三角形所在的近似曲面，然后根据近似曲面来得到交点处的切线向量 ss 和 ts，在 SetShadingGeometry 函数（见下面的代码）中使用 ss 和 ts 的叉积来得到法向量。

我们来看三角形求交的程序 Triangle::Intersect：

```
//暂时先把isect->shading.n初始化为跟isect->n同样的值
isect->n = isect->shading.n = Normal3f(Normalize(Cross(dp02, dp12)));
//如果三角mesh类里有顶点法向量数组或者切线向量
if (mesh->n || mesh->s){
    //根据三角形顶点向量或者切线向量来计算交点法向量
    ......
    // 根据光线微分得到的三角形着色几何dndu和dndv
    isect->SetShadingGeometry(ss, ts, dndu, dndv, true);
}
```

SetShadingGeometry 函数会根据 dndu 和 dndv 来计算着色法向量。

我们还记得，形状类的求交是 GeometricPrimitive::Intersect 调用的，调用完以后，该函数还检查几何法向量和着色法向量是否在同一个平面上：

```
CHECK_GE(Dot(isect->n, isect->shading.n), 0.);
```

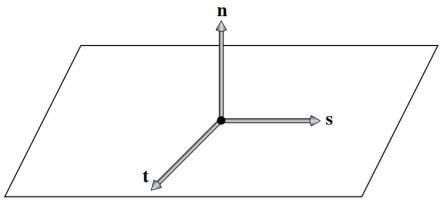
之后渲染积分器在调用计算散射函数程序时，该函数内会计算凹凸映射对着色法向量的影响：

```
//SamplerIntegrator::Li函数
isect.ComputeScatteringFunctions(ray, arena);
//Material::ComputeScatteringFunctions函数
if (bumpMap) Bump(bumpMap, si);//si是SurfaceInteraction对象
```

3 2　坐标转换

在第二章我们知道有着色坐标系，那么怎么将世界坐标系的向量变换到着色坐标系的上呢？

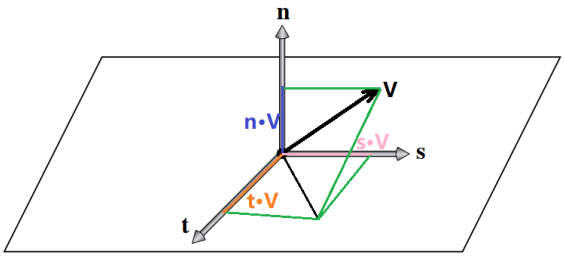
首先简单说一下什么是切线向量。对于空间中的曲面来说，一定有一个切面，过某点的切面与该点处的法向量垂直，而我们知道，一旦知道了平面法向量和平面上的任意一个向量 s，我们通过叉积就能得到平面上另外一个向量 t，该向量 t 与向量 s 垂直，t 与 s 就足够表示平面的空间朝向（再加上交点位置 p，s，t 和 p 就能得到平面的全部信息）。



如上图，着色空间的 s 轴在 BSDF 类中表示为 ss，t 轴表示为 ts。ss 直接来自于传入的切线值，然后将 ns 和 ss 叉积得到 ts。我们目前有了 ss,ts 和 n，设矩阵 M：

$$M = \begin{pmatrix} s_x & s_y & s_z \\ t_x & t_y & t_z \\ n_x & n_y & n_z \end{pmatrix} \tag{三.1}$$

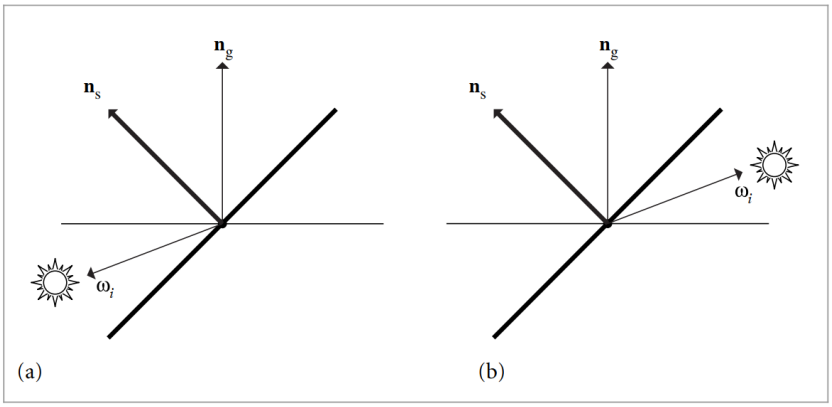
矩阵 M 乘以某个世界坐标系下的向量，就会把该向量变换到着色坐标系中，即 WorldToLocal 函数。这个矩阵其实非常好理解，其实就是投影：假如我们有一个世界坐标系下的向量 V：



则 V 在着色坐标系中的每个轴的坐标值就是它在每个轴的投影了，也就是它与轴的点乘值。注意因为矩阵 M 的行向量之间互相正交，列向量之间也互相正交，因此该矩阵的为正交矩阵，故矩阵的逆等于矩阵的转置。

LocalToWorld 函数就是从着色坐标系返回到世界坐标系中。

使用 ns 和 ng 会带来光泄露 (light leak) 或者暗点 (dark spot)。下图中的 (a)，ω_o 和 ω_i 不在表面的同侧，光应该是没有贡献的（除了透明物体），但如果这样计算的话就能计算到从 ω_i 方向入射的光，这样该点着色计算会过亮。下图 (b) 表示本来光能照到物体，但是着色表面不在 ω_i 的同半球内，因此该点着色计算会过暗。



PBRT 书中介绍了不错的解决方案，即根据 n_g 和 ω_i、ω_o 是否在同一个半球来决定在材质中使用 BRDF 还是 BTDF 来计算（注意这样有很大的好处，因为比如毛玻璃材质，它的 BRDF 和 BTDF 其实只是描述的方向不一样，其他内容是一样的，所以着色的结果在局部区域可以很好地过渡，不会出现亮点或者暗点）。

3 3 BSDF 内存管理

BSDF 在创建时使用的是内存管理类：

```
//ARENA_ALLOC是调用arena的宏
BSDF *b = ARENA_ALLOC(arena, BSDF);
BxDF *lam = ARENA_ALLOC(arena, LambertianReflection)(Spectrum(0.5f));
```

《误差界定和内存管理》一书中，我们并没有移植内存管理类，因此我们可以这么创建 BSDF：

```
BSDF *b = new BSDF;
BxDF *lam = new LambertianReflection(Spectrum(0.5f));
```

但这样非常容易造成内存泄漏，由于 BSDF 是求交后再创建的，所以每帧求交都会泄漏大量内存，造成渲染中未释放内存的累积。所以我们移植时用智能指针来管理（等您已经掌握了 PBRT 的所有主干内容后，可以自己动手移植一下内存管理类）。

3 4 Material 类

材料基类的定义非常简单，这里直接粘贴上来：

```
// TransportMode Declarations
enum class TransportMode { Radiance, Importance };
// Material Declarations
class Material {
  public:
    // Material Interface
    virtual void ComputeScatteringFunctions(SurfaceInteraction *si,
                                            MemoryArena &arena,
                                            TransportMode mode,
                                            bool allowMultipleLobes)
                                               const = 0;
    virtual ~Material();
    static void Bump(const std::shared_ptr<Texture<Float>> &d,
                     SurfaceInteraction *si);
};
```

材料基类里并没有直接定义 BxDF，这是因为不同的材料有不同的 BxDF 类型。TransportMode 表示表面相交点是从相机开始的还是从光源开始的，在双向路径追踪中有作用（我们需要知道光路径和相机路径），TransportMode 有两种，Radiance 表示是从光源发出的，Importance 表示是从相机发出的（涉及双向路径追踪的重要性采样），这里暂时不用管它。allowMultipleLobes 表示如果可以聚集的话，是否把多种类型的 BxDF 聚集到一个 BxDF 里（我们暂时不需要了解它）。

四　纹理与材质基类的移植

本来纹理类并不是这本书的要求，但是我们不得不掌握一下纹理基类和简单的派生类的构建，否则材质类无法实现。

texture 可以理解为是一种映射，将模型表面的点映射到其他空间的值，比如表面 (u,v) 索引映射到图像中的颜色值。纹理可能是高频变化的，但是相机的分辨率有限，因此需要反混淆技术。纹理的反混淆技术与光线微分息息相关，我们暂时只使用常量纹理（整个物体的纹理都是同一种颜色），因此不用在意反混淆技术。

对于常量纹理来说，不需要使用 (u,v) 坐标采样，这是最简单的纹理。

等讲纹理之后，我会专门用一本小书来讲怎么移植使用 assimp 库，assimp 库是一个专门解析模型文件的库，再加上 stb_image 库，读者就能渲染各种更好看的场景了。

4 1　纹理基类和常量纹理

纹理基类定义在 core 文件夹下的 texture.h 和 texture.cpp 文件里：

```
template <typename T>
class Texture {
    public:
    // Texture Interface
    virtual T Evaluate(const SurfaceInteraction &) const = 0;
    virtual ~Texture() {}
};
```

Evaluate 返回纹理值。这里的纹理类型表示为 T，PBRT 支持两种纹理类型，一种是 Float，另一种是 Spectrum。

对于常量纹理，定义在 textures 文件夹下的 constant.h 和 constant.cpp 文件里：

```
template <typename T>
class ConstantTexture : public Texture<T> {
    public:
    // ConstantTexture Public Methods
    ConstantTexture(const T &value) : value(value) {}
    T Evaluate(const SurfaceInteraction &) const { return value; }
    private:
    T value;
};
```

4 2　纹理类和材质基类的移植

纹理基类和常量纹理类我们直接复制到自己的系统里就好了。CreateConstantSpectrumTexture 和 CreateConstantFloatTexture 我们不需要移植，使用的时候直接用 new 创建纹理对象即可。

对于材质基类，构造函数里我们去掉了 MemoryArena 对象，并将凹凸贴图的函数 Bump 给去掉了：

```
// TransportMode Declarations
enum class TransportMode { Radiance, Importance };
// Material Declarations
class Material {
    public:
    // Material Interface
```

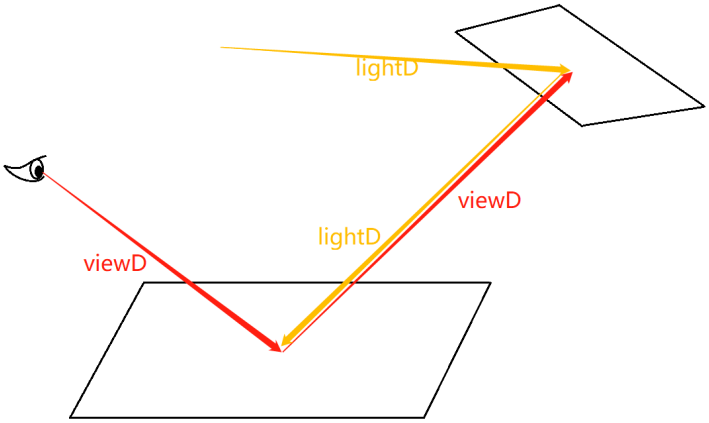
```
    virtual void ComputeScatteringFunctions(SurfaceInteraction *si,
                                            TransportMode mode,
                                            bool allowMultipleLobes) const =
                                                0;
    virtual ~Material() {}
};
```

另外再移植 reflection.h 和 reflection.cpp 文件里的类和函数，注意我们所有的移植都需要创建对应于 PBRT 的头文件和 cpp 文件。enum BxDFType 是需要直接复制过来的，class BxDF 也可以全部都复制过去。

在 reflection.h 文件里，所有求 cos，sin 和 tan 的函数全都移植过来，折射和反射也都移植过来，虽然我们还没有将折射和反射的计算，但是非常简单，反射就是根据一个入射方向求解出射方向，而折射还需要考虑是否会发生全反射，这些都是初高中的几何光学中最简单的知识了。SameHemisphere 的两个重载函数也都复制到自己的系统里。

4 3 wi 和 wo 的方向

本节会带着读者根据源码进行追踪和解读，请读者一定要完全弄懂讲述的全部内容（虽然非常简单）。为了更好的描述，我们按照如下建模：



设相机出发到表面的方向为 viewD，设灯光出发的方向到表面的方向为 lightD。对于多个表面来回反射的情况，一开始从相机方向发出的为 viewD，来自灯光方向的为 lightD。

以 Whitted 光线追踪为例，我们先到 integrator.h 和 integrator.cpp 文件定位到 SamplerIntegrator::Render 函数。在 do-while 循环语句中的 camera->GenerateRayDifferential 函数中，产生的 Ray 的方向是 viewD 方向。

之后进入 WhittedIntegrator 的 Li 函数，scene.Intersect 会计算 ray 与场景内物体的交点。但会在里面求出 SurfaceInteraction 对象的 wo 向量值。进入 shape 类的求交函数，我们选择 triangle.cpp 文件里的求交函数，可以看到求交中，对 SurfaceInteraction 对象赋值的代码是：*isect = SurfaceInteraction(...)，这里为 wo 赋的值为-ray.d。也就是说，wo 的方向是-viewD。

之后回到 WhittedIntegrator 的 Li 函数，isect 会调用 ComputeScatteringFunctions 来计算 BSDF。接着跟踪 Li 函数的程序，在遍历光源的 for 循环函数里，light->Sample_Li 会产生 wi 方向。我们找到 DiffuseAreaLight::Sample_Li 函数：

```
    *wi = Normalize(pShape.p - ref.p);
```

这里的 ref 表示的是 Interaction 对象，也就是 Ray 与物体表面的交点，pShape 表示的是光源上的采样点。因此 wi 方向表示的是-lightD 方向。第二章表示的的“着色几何 1”就是这个意思：wo 和 wi 的方向都是朝外的。

五　matte 材质

matte 材质是我们第一个要实现的材质。matte 表示的意思是无光泽的、粗糙的反射，也就是纯漫反射材料。在 PBRT 中，matte 可以分为两种，一种是经典的 Lambertian 漫反射，另一种是 OrenNayar 漫反射，我们只学习 Lambertian 漫反射模型，然后再学习 matte 材质，之后将该材质进行移植。

5 1　Lambertian 反射的原理和移植

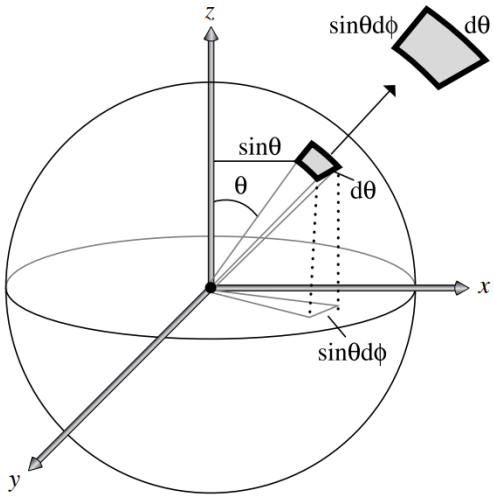
Lambertian 模型模拟的是完美的漫反射，即反射光在表面各个方向都是一样的。LambertianReflection 类定义在 reflection.h 和 reflection.cpp 文件里，该类的定义可以直接复制到我们的系统里，ToString 方法如果不需要的话可以去掉。

Spectrum R 我们可以认为表示的是纹理颜色，因为 BSDF 是求交以后生成的，因此材质类 Material 在调用 ComputeScatteringFunctions 计算生成 BxDF 时会赋给它纹理颜色值。（或者理解为，该值表示对入射光的反射和吸收比率，比如当 R 的值表示绿色的物体时，对于照射到该点的光束，反射的绿色成分会比较多，吸收的红和蓝色成分比较多）。

f(wo,wi) 函数表示的是光的分布，即从-wi 方向（注意前面说过，wi 和 wo 方向都是向外的）射入的光，射出到 wo 方向的光能量。

在理解光的分布时我们可以反着理解（入射 radiance 和出射 radiance 只是方向不一样）：从四面八方照射到表面的 L_i 值相等，但是单位表面接收到的辐射能量并不是 L_i，而是 $Li_icos\theta$，因为由第一章可以知道接收的 radiance 是单位垂直立体角下的辐射通量。因此反过来也一样，虽然 lambertian 的 BRDF 表示向任意方向反射的 L_i 都一样，但对于垂直于表面的反射方向中，单位立体角接收到的能量更多一些。

对于完美漫反射来说，f(wo,wi) 值为 R/π。即假如入射光的光照值的 RGB 表示是 $(1.0, 1.0, 1.0)$ 的白光，$R = (1.0, 0.2, 0.2)$ 表示该物体的颜色 (偏红色)，则反射到 wo 方向的光能量为 $(1.0, 0.2, 0.2)/\pi$。至于为什么是要除以 π，这是因为对立体角的半球积分：



如上图，我们知道，单位球体上的面积值 dA 等于对应立体角的值，$dA = sin\theta d\phi \cdot d\theta = d\omega$。

$$\int_\Omega f(\omega)cos(\theta)d\omega = \int_0^{2\pi}\int_0^{\pi/2} fcos\theta sin\theta d\theta d\phi = R \tag{五.1}$$

从而得到：$f = R/\pi$。

我们求解 lambertian 反射的渲染方程就是：

$$L_{\omega_o} = \int_\Omega \frac{Rcos(\theta)}{\pi} d\omega_i \tag{五.2}$$

5 2 Lambertian 采样和 Pdf

LambertianReflection 类继承了父类的方法：BxDF::Sample_f 和 BxDF::Pdf。为了更清楚的表示，我再写一下这个公式：

$$L_o = \int_\Omega f_r(p, \omega_o, \omega_i) L_i(p, \omega_i)(\omega_i \cdot n) d\omega_i \tag{五.3}$$

如何求积分呢？蒙特卡洛方法通过它的 BRDF 采样，对于 Lambertian 反射来说，反射到各个方向的概率都一样。在 Sample_f 采样方向时，就是根据二维随机数生成半球上的一个随机方向 wi。使用蒙特卡洛计算积分的时候：

$$L_o = \frac{1}{N} \sum_1^N \frac{f_r(p, \omega_o, \omega_i) L_i(p, \omega_i)(\omega_i \cdot n) d\omega_i}{p(\omega_i)} \tag{五.4}$$

其中 $p(\omega_i)$ 是我们使用的采样器的采样概率。

因此，由上一节中最后导出的对 lambertian 材质求解渲染方程的公式可以知道，在对 lambertian 材质进行采样随机方向的时候，采样密度要与 $cos\theta$ 成正比（我们甚至可以理解为 BRDF 把 $cos\theta$ 也包含了进去）。因此，在 BxDF::Sample_f 函数里：

```
1    //以与cos成正比的概率密度产生方向向量
2    *wi = CosineSampleHemisphere(u);
3    *pdf = Pdf(wo, *wi);
```

BxDF::Pdf 函数就是与 cos 值成正比：

```
1    Float BxDF::Pdf(const Vector3f &wo, const Vector3f &wi) const {
2        return SameHemisphere(wo, wi) ? AbsCosTheta(wi) * InvPi : 0;
3    }
```

这是根据下面的公式计算出来的。我们设 Pdf 与 $cos\theta$ 成正比关系，比例为 C：

$$\int_\Omega C cos(\theta) d\omega = \int_0^{2\pi} \int_0^{\pi/2} C cos\theta sin\theta d\theta d\phi = 1 \tag{五.5}$$

解得 $C = \frac{1}{\pi}$，因此对 $cos\theta$ 进行采样的 Pdf 就是 $\frac{cos\theta}{\pi}$。

5 3 产生随机方向

在 sampling.h 和 sampling.cpp 定义了很多生成随机方向的功能函数：

```
1    UniformSampleHemisphere
2    UniformHemispherePdf
3    UniformSampleCone
4    UniformSampleCone
5    UniformSampleDisk
6    ConcentricSampleDisk
7    CosineSampleHemisphere
8    CosineHemispherePdf
```

我们目前只用到了以 $cos\theta$ 成正比概率密度来产生方向的功能：CosineSampleHemisphere。在 [4] 中对随机方法已经讲得非常清楚了，为了防止大家遗忘，我简单把过程说一下。

我们假设在球面的纬度上生成角度为 ϕ 的概率密度是 $a(\phi)$，经度上生成角度为 θ 的概率密度是 $b(\theta)$。我们设随机方向与经度无关，也就是说，$a(\phi) = \frac{1}{2\pi}$，随机方向在任意经度上的可能性都一样，我们令随机

数 $r_1 \in [0,1)$ 表示发生在下面的概率。

$$r_1 = \int_0^{\phi} \frac{1}{2\pi} d\phi \tag{五.6}$$

解得：

$$\phi = 2\pi r_1 \tag{五.7}$$

而对于不同的纬度圈采样到某个点的概率是不一样的，找另一个随机数 $r_2 \in [0,1)$：

$$r_2 = \int_0^{\theta} 2\pi f(\theta) sin\theta d\theta \tag{五.8}$$

当均匀方向的时候，设 $f(\theta) = C$，概率密度在整个积分空间上的积分值为 1，因此可得（上式在书 [4] 上使用的是 $2\pi f(\theta) sin\theta$，其实由于 $f(\theta)$ 是个常数，因此可以把 $2\pi f(\theta)$ 这个整体看做是一个待求常数，不同的纬度线的长度比例就是 $sin\theta$，我们只需要注意这个比例）：

$$\int_0^{\pi} 2\pi C sin\theta d\theta = 1 \tag{五.9}$$

解得：$C = \frac{1}{4\pi}$。所以把 C 代入原式，得到：$cos(\theta) = 1 - 2r_2$

当 $f(\theta) = Ccos(\theta)$ 时，也就是前面所述的，随机方向的概率密度与 $cos(\theta)$ 成正比。代入进去就能得到 C 的值，然后把 C 代入到里面，就能得到：$cos(\theta) = \sqrt{1 - r_2}$。

之后再根据 ϕ 和 θ 值计算出 (x, y, z) 坐标值。

5 4 lambertian 的反射比

最后一个内容关于 Lambertian 漫反射的知识是反射比。根据第二章求反射比的公式，我们可以得到对于反射到单个方向的反射比：

$$\rho_{hd} = \int_{\Omega} \frac{R}{\pi} |cos\theta_i| d\omega_i = R \tag{五.10}$$

对于反射到全部方向的反射比：

$$\rho_{hd} = \int_{\Omega} (\int_{\Omega} \frac{R}{\pi} |cos\theta_i| d\omega_i) |cos\theta_o| d\omega_o = R \tag{五.11}$$

因此 LambertianReflection::rho 返回值都是 R。

5 5 材质类的移植和讲解

移植一共有两部分，一是 LambertianReflection 类和成员函数的移植，二是 sampling.h 和 sampling.cpp 里相关函数的移植。

BxDF::rho 方法我们也移植过去，虽然暂时用不到，但其实就是使用的蒙特卡罗方法来求解的反射比，等我们后面学习 BSSRDF 时会重新介绍反射比的用途和蒙特卡洛求解方法。BRDF 的移植非常简单，这里不再赘述。

SurfaceInteraction 类的另一个多参数的构造函数需要添加到我们的系统里。如果您之前在《误差界定和内存管理》书中移植了误差界定的内容，父类 Interaction 和该类都在上面加上参数 pError。SurfaceInteraction 该类加上成员变量：

```
1     BSDF *bsdf = nullptr;
2     Vector3f dpdu, dpdv;
3         Normal3f dndu, dndv;
4     struct {
5         Normal3f n;
6         Vector3f dpdu, dpdv;
7         Normal3f dndu, dndv;
8     } shading;
```

Triangle::Intersect 函数里我们要添加上：

```
        // Compute triangle partial derivatives
        Vector3f dpdu, dpdv;
        Point2f uv[3];
    //注意模型里没有纹理坐标也会进行赋值，详情见triangle.h头文件
        GetUVs(uv);

        // Compute deltas for triangle partial derivatives
        Vector2f duv02 = uv[0] - uv[2], duv12 = uv[1] - uv[2];
        Vector3f dp02 = p0 - p2, dp12 = p1 - p2;

        // 如果你的程序里移植了误差界定，就加上相关内容，没有的话就可以不用加。
    // ......

        Point2f uvHit = b0 * uv[0] + b1 * uv[1] + b2 * uv[2]; //重心插值的纹理坐标
        Point3f pHit = b0 * p0 + b1 * p1 + b2 * p2; //重心插值的击中点

        // Fill in _SurfaceInteraction_ from triangle hit
        *isect = SurfaceInteraction(pHit, pError, uvHit, -ray.d, dpdu, dpdv,
                Normal3f(0, 0, 0), Normal3f(0, 0, 0), ray.time,
                this, faceIndex);

        // Override surface normal in _isect_ for triangle
        isect->n = isect->shading.n = Normal3f(Normalize(Cross(dp02, dp12)));
    //不考虑法向量要反转
        //if (reverseOrientation ^ transformSwapsHandedness)
        //      isect->n = isect->shading.n = -isect->n;
```

然后我们移植 BSDF 类。该类可以直接全都拷贝到自己的系统上。但是里面有些理论知识并没有涉及到，所以只介绍源码的实现目的，具体为什么要这么做（比如 f、Pdf 的遍历循环）我们会在后面的书中详细解释原理。

BSDF::NumComponents 返回值表示该 BSDF 里某种类型的 BxDF 有多少个。BSDF::ToString 方法可以根据需要保留或者舍去。BSDF::f 方法，BSDF::Sample_f 方法，BSDF::Pdf 方法和两个重载的 BSDF::rho 方法都比较麻烦，我们一个一个讲解。

BSDF::rho 最简单，就是遍历，然后返回与参数 BxDFType flags 相同的 BxDF 反射模型的 rho。

BSDF::f 函数中，bool reflect 表示入射光和反射光是否在同一个半球内。之后遍历 BxDF 数组，寻找与 flag 相同类型的 BxDF，如果在半球内且类型为 BSDF_REFLECTION，则正确，否则如果不在同一个半球内且类型为 BSDF_TRANSMISSION，则也是判断正确，这样就令 Spectrum f 加上 bxdfs[i]->f(wo, wi)。最后返回 f。

BSDF::Sample_f 最麻烦，首先 matchingComps 表示传入参数 sampledType 类型的 BxDF 的数量。如果没有，就给类型赋值为 0，返回黑色。std::floor 向下取整：

```
    //比如matchingComps = 4，那么根据随机数u[0]，comp得到的值就在0-3之间。
    int comp =
        std::min((int)std::floor(u[0] * matchingComps), matchingComps - 1);
```

之后再循环 nBxDFs 的过程中，选择第 comp 个 sampledType 类型的 BxDF 为采样用的 BxDF。之后计算 uRemapped 表示复用随机数 u[0]，产生的二维随机数用于采样 BxDF：调用 bxdf->Sample_f。如果采样后得到的 pdf 为 0，则返回黑色（说明有可能是完美的镜面光，需要其他采样方式）。之后遍历区间里所有相同类型的 BxDF，将 PDF 都计算并加起来除以总数，得到平均的 PDF。然后就是相当于调用了 BSDF::f 方法，计算反射到 wo 方向的光分布。

本书中用到的 BSDF 只包含一个 LambertianReflection，所以就相当于 BSDF 直接使用了 LambertianReflection 的 BxDF。

我们需要手动释放内存（但我建议改成使用智能指针的方法，这样虽然慢，但是可以保证内存可以被及时释放干净，或者移植和使用 PBRT 的内存管理方法）：

```
BSDF::~BSDF() {
    for (int i = 0; i < nBxDFs; i++)
        bxdfs[i]->~BxDF();
}
SurfaceInteraction::~SurfaceInteraction(){
    if(bsdf)
        bsdf->~BSDF();
}
```

接下来我们移植 MatteMaterial 类。尽管我们没有凹凸贴图（bump），也没有讲 OrenNayar 材质（与纹理参数 sigma 有关），但因为它们都是纹理对象，所以我们就暂时带着它们。ComputeScatteringFunctions 函数我们不需要参数 arena，我们移植到自己系统的函数如下：

```
// MatteMaterial Method Definitions
void MatteMaterial::ComputeScatteringFunctions(SurfaceInteraction *si,
                                               TransportMode mode,
                                               bool allowMultipleLobes)
                                                  const {
    // Evaluate textures for _MatteMaterial_ material and allocate BRDF
    si->bsdf = new BSDF(*si);
    Spectrum r = Kd->Evaluate(*si).Clamp();
    float sig = Clamp(sigma->Evaluate(*si), 0, 90);
    if (!r.IsBlack()) {
        if (sig == 0)
            si->bsdf->Add(new LambertianReflection(r));
    }
}
```

如果全部正确编译成功，就说明基础功能已经基本完备，下一章我们把整个系统流程整合好，并渲染出一幅图像。

六　全部功能整合与使用流程

回顾一下我们的系统已经具备了哪些内容呢？形状类，加速器结构，颜色与光谱类，相机类，采样器类，BRDF 类。如果你能不看源码就想得出着色的计算流程，那说明你已经理解的很好了。让我们回顾一下：首先，表面交点对象 isect 调用 ComputeScatteringFunctions，它调用的是基元 primitive 的同名函数，并调用 material 对象的同名函数。之后表面交点对象 isect 便得到了 bsdf 对象，并根据 bsdf 进行计算。

6 1　计算散射函数的移植

首先为 Primitive 基类和 GeometricPrimitive 与 Aggregate 派生类声明函数，其中 Primitive 类应该声明为纯虚函数。

```
1        virtual void ComputeScatteringFunctions(SurfaceInteraction *isect,
2                TransportMode mode,
3                bool allowMultipleLobes) const;
```

Aggregate 类无需实现具体的计算散射的函数，因为它只是个加速结构。GeometricPrimitive 类加入成员对象：

```
1        std::shared_ptr<Material> material;
```

```
1    void GeometricPrimitive::ComputeScatteringFunctions(
2        SurfaceInteraction *isect, TransportMode mode,
3        bool allowMultipleLobes) const {
4        if (material)
5            material->ComputeScatteringFunctions(isect, mode,
6                allowMultipleLobes);
7        //CHECK_GE(Dot(isect->n, isect->shading.n), 0.);
8    }
```

在 SurfaceInteraction 函数中也要实现同名方法：

```
1    void SurfaceInteraction::ComputeScatteringFunctions(const Ray &ray,
2        bool allowMultipleLobes,
3        TransportMode mode) {
4        primitive->ComputeScatteringFunctions(this, mode,
5            allowMultipleLobes);
6    }
```

GeometricPrimitive 的构造函数也得改一下：

```
1    GeometricPrimitive::GeometricPrimitive(const std::shared_ptr<Shape> &
       shape,
2        const std::shared_ptr<Material> &material)
3        : shape(shape), material(material){
4        primitiveMemory += sizeof(*this);
5    }
```

在 GeometricPrimitive::Intersect 求交函数里应该加上下面这行代码，表示交点的基元类是当前基元。

```
1    isect->primitive = this;
```

6 2 材质的实现

我们写如下代码，实现纹理和材质（我们渲染的内容包含了一个地板和一条龙）：

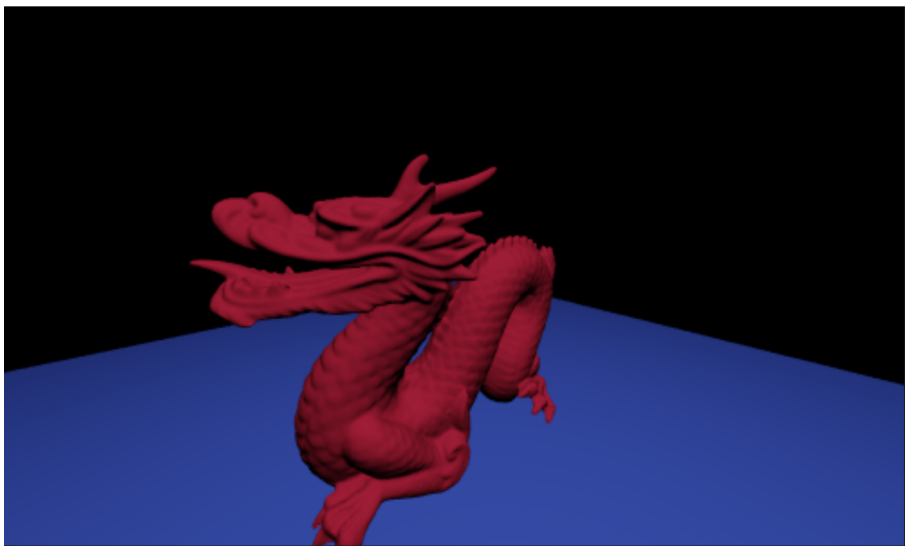
```
        //纹理
        Spectrum floorColor; floorColor[0] = 0.2; floorColor[1] = 0.3;
            floorColor[2] = 0.9;
        Spectrum dragonColor; dragonColor[0] = 0.8; dragonColor[1] = 0.1;
            dragonColor[2] = 0.2;
        std::shared_ptr<Texture<Spectrum>> KdDragon = std::make_shared<
            ConstantTexture<Spectrum>>(dragonColor);
        std::shared_ptr<Texture<Spectrum>> KdFloor = std::make_shared<
            ConstantTexture<Spectrum>>(floorColor);
        std::shared_ptr<Texture<float>> sigma = std::make_shared<
            ConstantTexture<float>>(0.0f);
        std::shared_ptr<Texture<float>> bumpMap = std::make_shared<
            ConstantTexture<float>>(0.0f);
        //材质
        std::shared_ptr<Material> dragonMaterial = std::make_shared<
            MatteMaterial>(KdDragon, sigma, bumpMap);
        std::shared_ptr<Material> floorMaterial= std::make_shared<
            MatteMaterial>(KdFloor, sigma, bumpMap);
        //将物体填充到基元
        for (int i = 0; i < plyi->nTriangles; ++i)
                prims.push_back(std::make_shared<GeometricPrimitive>(tris[i
                    ], dragonMaterial));
        for (int i = 0; i < nTrianglesFloor; ++i)
                prims.push_back(std::make_shared<GeometricPrimitive>(
                    trisFloor[i], floorMaterial));
```

之后我们在渲染函数里这么做：

```
    do {
        CameraSample cs;
        cs = pixel_sampler->GetCameraSample(pixel);
        Ray r;
        camera->GenerateRay(cs, &r);
        SurfaceInteraction isect;
        if (scene.Intersect(r, &isect)) {
            //计算散射
            isect.ComputeScatteringFunctions(r);
            // 对于漫反射材质来说，wo不会影响后面的结果
            Vector3f wo = isect.wo;
            Vector3f LightNorm = Light - isect.p;
            LightNorm = Normalize(LightNorm);
            Vector3f wi = LightNorm;
            Spectrum f = isect.bsdf->f(wo, wi);
            float pdf = isect.bsdf->Pdf(wo, wi);
            //乘以3.0的意义是为了不让图像过暗
            colObj += pdf*f*3.0f;
```

```
        }
    } while (pixel_sampler->StartNextSample());
    colObj = colObj / (float)pixel_sampler->samplesPerPixel;
```

得到渲染结果如下：

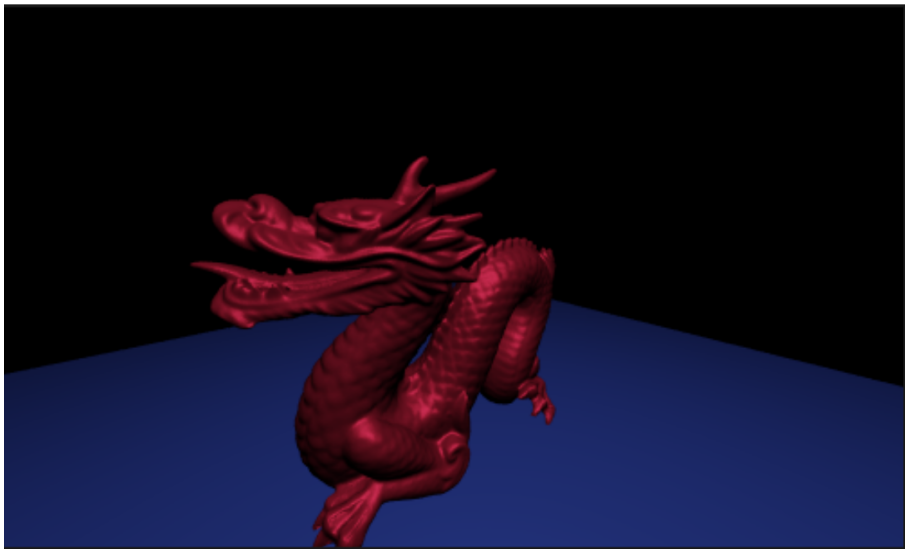


因为我们缺少面光源，我们的采样方向也是根据点光源位置计算得到的，因此采样 Pdf 我们暂时无法计算出（虽然其实就是 1.0），所以为了渲染出具有一定光分布规律的图像，我们这里使用 pdf*f 来造成色差，另外我们忽视了点光源的随距离衰减的特性。这只是为了使用本书中移植过的多个类来渲染一个结果。

我们甚至还可以再乘以一个 Blinn-Phong 照明的系数，让渲染结果更好看一些：

```
    Vector3f viewInv = -r.d;
    Vector3f H = Normalize(viewInv + LightNorm);
    float Ls = Dot(H, isect.n); Ls = (Ls > 0.0f)? Ls:0.0f;
    Ls = pow(Ls, 32);
    float Ld = Dot(LightNorm, isect.n); Ld = (Ld > 0.0f) ? Ld : 0.0f;
    //乘以1.5倍是因为太暗了
    float Li = 1.5 * (0.2 + 0.2 * Ld + 0.7*Ls);
    colObj += Li * pdf*f*3.0f;
```

渲染结果如下：



七　本书结语

本文难点在于怎么剥离出不需要的内容，然后实现一个包含简单的材质和纹理的渲染流程。本文从 3 月 4 日开始，3 月 8 日完成。

现在，距离完整的光线追踪器，我们还剩下最后一个内容——光源，因此下一本书我会介绍光源接口，包括形状类和面光源的集成和表示。

移植完光源系统以后我们就可以实现一个真正意义上的基于 PBRT 的 Whitted 光线追踪引擎了，因此下一本书我们移植完光源以后，就开始着手实现它。

参考文献

[1] Pharr M, Jakob W, Humphreys G. Physically based rendering: From theory to implementation[M]. Morgan Kaufmann, 2016.

[2] Shirley P. Ray Tracing in One Weekend[J]. 2016.

[3] Shirley P. Ray Tracing The Next Week[J]. 2016.

[4] Shirley P. Ray Tracing The Rest Of Your Life[J]. 2016.