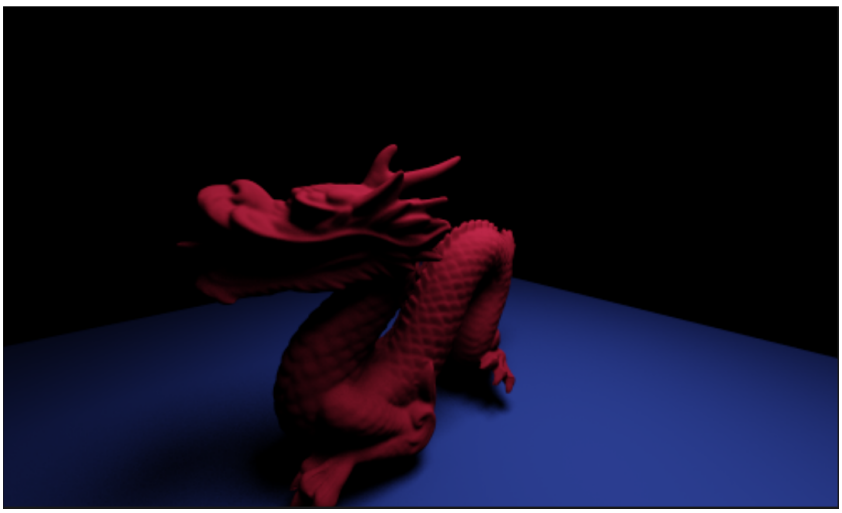
PBRT 系列 9-代码实战-灯光基础与完整的光线追踪器

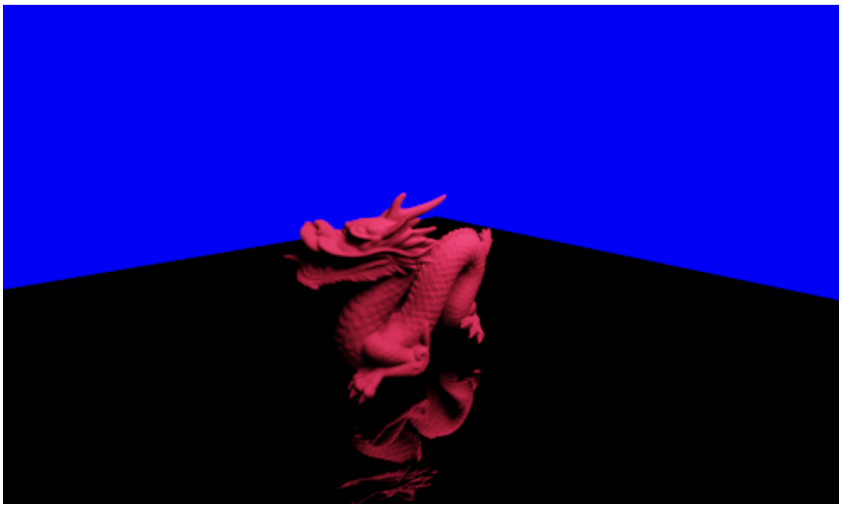
Dezeming Family

2021 年 3 月 8 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 的光源系统。移植点光源和面光源到自己的引擎上。学习和移植 PBRT 的完美镜面物体。实现 Whitted 光线追踪器到我们的系统中。渲染出下面的图像：





本文于 2022 年 7 月 11 日进行再版，提供了源码。注意图形 GUI 界面和本文中展示的有点区别，但并不影响学习。源码见网址 [https://github.com/feimos32/PBRT3-DezemingFamily]。

前言

万事俱备，只欠光源。其实，我们在移植 PBRT 的时候，可以将全部的基类都移植上去，然后慢慢补充功能，这样也会省得删除一些不必要的内容，但我们选择了另外一种方式——逐步添加各个基类。

我们要完全实现 PBRT 的 Whitted 光线追踪，而不是之前《Whitted 光线追踪引擎》上在 [2][3][4] 上构建的光追器，我们还需要最后一个组件：面光源。我们首先学习和移植 PBRT 的光源系统，实现简单的点光源和面光源。然后就动手实现 PBRT 中 Whitted 光线追踪器。

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　PBRT 的光源描述

本章描述 PBRT 中的光源，介绍光源的理解，PBRT 光源类的接口。

1 1　光源物理描述

所有温度高于绝对零度的物体都有运动的原子。正如麦克斯韦方程所描述的那样，携带电荷的原子粒子的运动会导致物体在一定波长范围内发射电磁辐射。对于室温下的物体来说，大部分发射都是在红外频率下进行的；物体需要加热升温才能发射在可见光频率下的电磁辐射。

人们发明了许多不同类型的光源来将能量转换成发出的电磁辐射。理解其中的一些物理过程有助于精确地为渲染光源建模。

白炽灯有一根小钨丝。通过灯丝的电流加热灯丝，进而使灯丝发出电磁辐射，其波长分布取决于灯丝的温度。外层的磨砂玻璃外壳通常用于吸收产生的一些波长，以让我们获得所需的 SPD。使用白炽灯时，发出的电磁辐射的 SPD 中的大部分能量都在红外波段，这反过来意味着消耗的大部分能量都转化为热而不是光。

卤素灯也有钨丝，但周围的外壳充满了卤素气体。随着时间的推移，白炽灯中的部分灯丝在加热时会蒸发；卤素气体会使蒸发的钨返回灯丝，从而延长灯的寿命。因为它回到灯丝上，蒸发的钨不会附着在灯泡表面（就像普通白炽灯泡那样），这也可以防止灯泡变暗。

气体放电灯使电流通过氢、氖、氩或汽化的金属气体，从而使光以特定波长发射，具体波长取决于气体中的特定原子。（选择在红外频率中发射相对较少电磁辐射的原子作为气体。）因为较宽的波长光谱通常比选择的某种原子直接产生的光谱在视觉上更理想，灯泡内部的荧光涂层通常用于将发射频率转换到更宽的范围。（荧光涂层也有助于将紫外线波长转换为可见光波长。）

LED 灯是基于电发光的：它们使用的材料由于电流通过而发射光子。

对于所有这些光源，基本的物理过程是电子与原子碰撞，从而将它们的外层电子推向更高的能级。当这样的电子返回到较低的能级时，就会发射光子。还有许多其他有趣的过程可以产生光，包括化学发光（如在荧光棒中看到的）和生物发光——比如萤火虫的化学发光形式。尽管它们本身很有趣，但我们在这里不进一步讨论它们的机制。发光效率衡量光源如何有效地将功率转换为可见光照明，这说明对于人类观察者来说，不可见波长的发射几乎没有价值。发光效率是 photometric quantity（emitted luminous flux）与 radiometric quantity （它使用的总功率或它发射总波长的总功率，以 flux 测量）的比值：

$$\frac{\int \Phi_e(\lambda)V(\lambda)d\lambda}{\int \Phi_i(\lambda)d\lambda} \tag{一.1}$$

$V(\lambda)$ 可以理解为只在可见光的范围不为 0 的函数（人眼对光反应的曲线）。

PBRT 中定义了黑体（blackbody），我们目前只作为了解。黑体是一个完美的电磁辐射发射器：它能在物理上尽可能有效地将能量转化为电磁辐射。虽然真正的黑体在物理上是不可实现的，但有些发射器表现出接近黑体的行为。黑体也有一个有用的闭式表达，他们的发射是温度和波长的函数，这是非常有用的建模非黑体发射器。

黑体之所以这样命名，是因为它们吸收了所有的入射能量，却没有反射任何一种能量。因此，无论有多少光照亮它，一个实际的黑体将出现完美的黑色。直观地说，完美吸收体也是完美发射体，因为吸收是发射的逆操作。因此，如果时间倒转，所有完全吸收的能量将完全有效地重新发射出去。普朗克定律给出了黑体发射的辐射强度，它是波长　和温度 T 的函数，测量单位为开尔文。黑体发光在 PBRT 中的 spectrum.h 中有定义，并用在了 Hosek 天空模型里（可以用来大致预测类似地球的行星，即大小和大气组成相似的行星上的户外场景）。

我们的其他光源，比如面光源。就不是黑体光源，也就是说，如果有其他光源照射到面光源上，会把面光源照得更亮。

国际照明委员会（CIE）定义了许多“标准光源”，标准光源 A（Standard Illuminant A）于 1931 年引入，旨在代表普通白炽灯。它对应于一个大约 2856K 的黑体辐射器（它最初被定义为 2850K 的黑体，但普朗克定律中所用常数的精度后来提高了）。B 和 C 光源旨在模拟一天中两个时间段的日光，并由 A 光源与特定滤光片组合产生，它们不再被使用。E 光源定义为常量值 SPD，仅用于与其他光源进行比较。

1 2 光源接口

光源定义在 core 文件夹下的 light.h 和 light.cpp 文件里。光源基类的构造函数有四个参数：

```
Light(int flags, const Transform &LightToWorld,
    const MediumInterface &mediumInterface, int nSamples = 1);
```

第一个参数表示光源类型，我们有四种光源大类，定义在 LightFlags 枚举类里，比如点光源，方向光，面光源和无限面光源。第二个参数表示光源变换到世界坐标系的变换。第三个参数表示参与介质与光源的关系。即光源可能身处参与介质中。第四个参数 nSamples 用于面光源，其中可能需要将多个阴影光线跟踪到灯光以计算软阴影，默认光源采样数为 1。

我们把 LightFlags 枚举类、IsDeltaLight 函数和去掉 MediumInterface 的 Light 类都移植到我们的系统中。因为我们还没有学习和实现光线微分，所以 Light::Le 方法的光线微分参数改为 Ray。

```
    virtual Spectrum Le(const RayDifferential &r) const { return Spectrum(0.
        f); }
```

Light::Sample_Li() 返回 Interaction 交点发射到光源采样点的向量 ω_i（注意，在《反射与材质》中我们说过，ω_i 虽然按理来说表示入射光方向，但是是要让它朝向外侧的），其中参数 VisibilityTester 是用来做可见性测试的，防止灯光被某些东西挡到。

所有灯光还必须能够返回其总发射功率，此值对于可能希望为场景中贡献最大的灯光投入额外计算资源的光传输算法非常有用。因为在某些地方不需要发射功率的精确值，所以许多这种方法的实现将计算该值的近似值，而不是花费计算精力来找到精确值。

Light::Preprocess() 方法在渲染之前调用。它包含场景作为参数，以便光源可以在渲染开始之前确定场景的特性。默认实现为空，但某些实现（例如 DistantLight）使用它来记录场景范围的界定区间。

1 3 可见性测试的移植

VisibilityTester 允许灯光在参考点和光源相互可见的假设下返回辐射值。然后，积分器可以在产生阴影光线的之前决定来自入射方向的照明是否相关，例如，非半透明面光源的背面的光对从另一侧没有任何贡献。

可见性测试 VisibilityTester 可以完全把定义移植到自己的系统上。我们需要实现里面的两个函数：Unoccluded 和 Tr。Tr 函数需要考虑参与介质对光的衰减，我们还没有实现参与介质，因此我们并不真的实现这个函数。

```
    Spectrum Tr(const Scene &scene, Sampler &sampler) const { return
        Spectrum(0.f); }
```

Unoccluded 函数不考虑参与介质，只返回是否光源被阻挡。

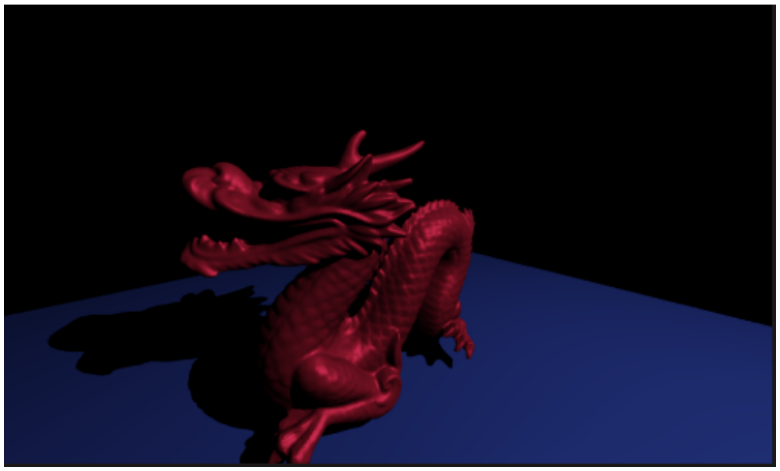
```
    bool VisibilityTester::Unoccluded(const Scene &scene) const {
        return !scene.IntersectP(p0.SpawnRayTo(p1));
    }
```

Interaction 类的 SpawnRay 函数、SpawnRayTo 函数我们移植到 Interaction 类里，之前我们没有用到，只是学习了一下关于 Ray 的误差界定。注意 GetMedium(d) 我们还没有参与介质，因此去掉这个参数（我们 Ray 的构造函数中也没有这个参数）。SpawnRayTo 函数输入是另一个交点类，之后会生成 Ray，出发点在 p0 交点，方向是 p0-p1 的方向，没有经过单位化，最大值设为了 1-0.00001，这样就相当于将 Ray 的相交计算限定在了交点 p0 到光源采样点 p1 之间了。Interaction 还有很多重载的构造函数，我们也都移植到自己的系统里。

我们还没有具体的灯光类，但是我们可以先测试一下光源函数：

```
    //光源位置
        Point3f LightPosition(1.0, 4.5, -6.0);
    //计算对光源是否可见并着色
    SurfaceInteraction isect;
    if (scene.Intersect(r, &isect)) {
        //交点类p1的交点设为光源位置
        Interaction p1; p1.p = LightPosition;
        VisibilityTester t(isect, p1);
        if(t.Unoccluded(scene)){
            //计算着色
            ......
        }
    }
```

得到渲染结果：



可以看到龙的左边产生了阴影，说明相交测试类移植成功了。

二　具体的光源类型

本章我们介绍两种光源：点光源和面光源。点光源会延伸出“手电筒光”和纹理投射光，但我们在这本书里不考虑这些内容（尽管手电筒非常简单）。点光源我们其实已经在上一章中实现了，只不过没有定义在光源接口里，现在我们定义在光源接口里。

2 1　点光源

点光源类定义在 light 文件夹的 point.h 文件和 point.cpp 文件里。

点光源的方法都非常简单和好理解，点光源默认在自身坐标系的 (0,0,0) 点，通过变换矩阵变换到世界坐标系位置。对于点光源来说，能量随距离的衰减成平方关系。

Point 类的移植还是去掉参与介质的接口就好了。PointLight::Sample_Li 函数中，参数 pdf 设置为 1，因为在光源上采样，只会采样到光源所在点，可能性就是 1。PointLight::Sample_Le 函数表示的是从光源进行方向的采样，我们可以看到，点光源是从球上均匀采样的，位置 pdf 是 1，方向 pdf 就是球均匀采样方向的概率密度。PointLight::Pdf_Li 表示当我们给定一个当前交点（第一个参数）和光源的发射方向（第二个参数）时，与光源方向立体角构成的概率密度。但我们知道，点光源非常小，我们传入的任何方向都无法击中点光源，因此返回 0。PointLight::Pdf_Le 是用在双向光传输算法里的（BDPT,PhotonMap），我们虽然当前移植到了自己的系统里，但目前我们不讲解它的用途。

2 2　点光源的使用测试

为什么不直接讲完面光源再进行使用和测试，是因为我们前面有些涉及到光源的内容都剥离掉了，本节我们一点一点补充回来。在补充的过程中也会对光源的使用流程更加清楚，为面光源做铺垫（面光源相对比较麻烦）。

首先，Scene 类的构造函数需要修改成与 PBRT 一致的函数，只是我们没有无限环境光，所以注释掉里面添加环境光的内容。

在我们构建场景的函数中：

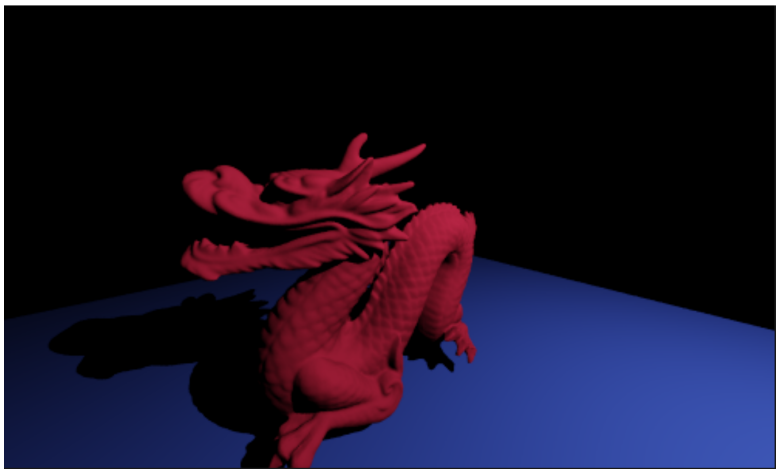
```
1    //构造光源，30.0是为了防止场景太暗。
2        Spectrum LightI(30.f);
3        Transform LightToWorld; LightToWorld = Translate(Vector3f(1.0f, 4.5f
             , -6.0f))*LightToWorld;
4        std::shared_ptr<Light> pointLight = std::make_shared<PointLight>(
             LightToWorld, LightI);
5        std::vector<std::shared_ptr<Light>> lights;
6        lights.push_back(pointLight);
7    //构造场景。prims是基元数组（见《形状和加速器》一书）。
8    worldScene = std::make_unique<Scene>(std::make_shared<BVHAccel>(prims,
         1), lights);
```

我们需要对渲染积分器的 Render 函数进行幅度比较大的修改，因此我把完整代码放在下面：

```
1  //渲染图像尺寸：RasterWidth X RasterHeight
2  for (int j = 0; j < RasterHeight; j++) {
3      for (int i = 0; i < RasterWidth; i++) {
4          //像素索引
5          int offset = (i + RasterWidth * j);
6          std::unique_ptr<Sampler> pixel_sampler = sampler->Clone(offset);
7          Point2i pixel(i, j);
8          //采样器开始工作
```

```
9           pixel_sampler->StartPixel(pixel);
10          Spectrum colObj(0.0f);
11          do {
12              CameraSample cs;
13              cs = pixel_sampler->GetCameraSample(pixel);
14              Ray r;
15              camera->GenerateRay(cs, &r);
16              SurfaceInteraction isect;
17              if (scene.Intersect(r, &isect)) {
18                  VisibilityTester vist;
19                  Vector3f wi;
20                  Interaction p1;
21                  float pdf_light;//采样光的Pdf
22                  //采样光
23                  Spectrum Li = scene.lights[0]->Sample_Li(isect,
                        pixel_sampler->Get2D(), &wi, &pdf_light, &vist);
24                  //如果光源没有被遮挡
25                  if(vist.Unoccluded(scene)){
26                      isect.ComputeScatteringFunctions(r);
27                      Vector3f wo = isect.wo;
28                      //采样散射光分布函数
29                      Spectrum f = isect.bsdf->f(wo, wi);
30                      //散射Pdf
31                      float pdf_scattering = isect.bsdf->Pdf(wo, wi);
32                      //渲染积分
33                      colObj += Li * pdf_scattering * f *3.0f / pdf_light;
34                  }
35              }
36          } while (pixel_sampler->StartNextSample());
37          colObj = colObj / (float)pixel_sampler->samplesPerPixel;
38          //保存该颜色值值到缓冲区
39          ......
40      }
41  }
```

渲染得到如下结果：



2 3　面光源

面光源需要很多内容，我们先从开始定义基类开始。面光源基类定义在 light.h 中，我们直接移植到自己的系统里。注意去掉 MediumInterface 参数，我们保留记录面光源数量的参数：

```
1    static long long numAreaLights = 0;
```

然后是它的派生类：DiffuseAreaLight。该类定义在 light 文件夹下的 diffuse.h 和 diffuse.cpp 文件中。该类成员变量 twoSided 表示光是否可以从 Shape 的两边发出，为 false 表示只从法向量的一遍发出。nSamples 参数表示光源上需要采样几个点，比如在 DirectLightingIntegrator 积分器中会用到它。该类里面的 Sample_Le 方法和 Pdf_Le 方法比较复杂，我们暂时不实现里面的代码，将函数体保留，并注释掉里面的内容（Sample_Le 给一个简单的返回值即可）。该类别的方法都需要进行移植。

下面两行分别在两个函数里出现的代码我们都没有实现过：

```
1    //Sample_Li
2    Interaction pShape = shape->Sample(ref, u, pdf);
3    //Pdf_Li
4    shape->Pdf(ref, wi);
```

shape->Sample 函数有两个重载函数，一个是子类实现的采样函数，例如三角形类；另一个是基类实现的采样函数：

```
1    //子类函数
2    Sample(const Point2f &u, Float *pdf)
3    //基类函数
4    Sample(const Interaction &ref, const Point2f &u,
5                                Float *pdf) const;
```

还有两个 Pdf 函数，其中一个是返回采样点在表面的概率密度，其实就是 1 除以面积。

另一个 Pdf 函数的意义是给一个交点对象，给它一个方向，判断这个交点向该方向发出的 Ray 是不是与本面光源相交，如果相交了就计算对光采样的概率密度函数（见 [4] 中的直接采样光的一节）。注意 Ray ray = ref.SpawnRay(wi); 的意义我们在《误差界定和内存管理》提到过，是根据当前点 p 的位置和误差界定发出新的光线 Ray，防止新产生的 Ray 与自身交点所在表面再次相交。

我们先实现 Shape 子类的 Sample 函数，再去实现基类的 Sample 函数。还好我们只比较完整地实现了三角形，因此移植起来非常容易。

Triangle::Sample 函数首先使用 UniformSampleTriangle 采样到随机位置，该位置表示的是针对顶点的偏移。然后映射到实际三角形的顶点上。然后计算三角形法向量。值得一提的是，就算我们的模型提供了法向量数组，在计算的时候也只用来判断根据顶点坐标计算的法向量方向。输入参数的 pdf 赋值为 1 除以面积。而在 Shape::Sample 中，会将 Pdf 再计算为与立体角相关的对光采样的概率密度（参考 [4] 中的对光采样）。

Sphere::Sample 的代码很长，鉴于以后我不再打算使用球体作为模型（何况球体是可以面片化的），因此我就不再填充球类的方法了，随便返回个值应付过去（如果你想完全实现一遍 PBRT 的球体其实很简单，就是很麻烦，我们的思想仍旧是：人生苦短，及时行乐）。

```
1    Interaction Sphere::Sample(const Point2f &u, float *pdf) const {
2        Interaction it;
3        return it;
4    }
```

Shape::Sample 函数非常简单，注意 std::isinf 表示判断是否为非法数（例如除以 0 得到的数）。

移植完以后编译通过，就说明没有问题了。

2 4 面光源的使用测试

场景文件加载过程中，在 api.cpp 文件里 pbrtShape 函数中构建面光源：

```
    for (auto s : shapes) {
        // Possibly create area light for shape
        std::shared_ptr<AreaLight> area;
        if (graphicsState.areaLight != "") {
            area = MakeAreaLight(graphicsState.areaLight, curTransform[0],
                                 mi, graphicsState.areaLightParams, s);
            if (area) areaLights.push_back(area);
        }
        prims.push_back(
            std::make_shared<GeometricPrimitive>(s, mtl, area, mi));
    }
```

因此 GeometricPrimitive 的构造函数我们需要进行修改。我们在该类中添加第三个成员变量：

```
    std::shared_ptr<AreaLight> areaLight;
```

并修改它的构造函数（目前该类仅剩最后一个参与介质成员没有添加了）。

我们开始构建场景：

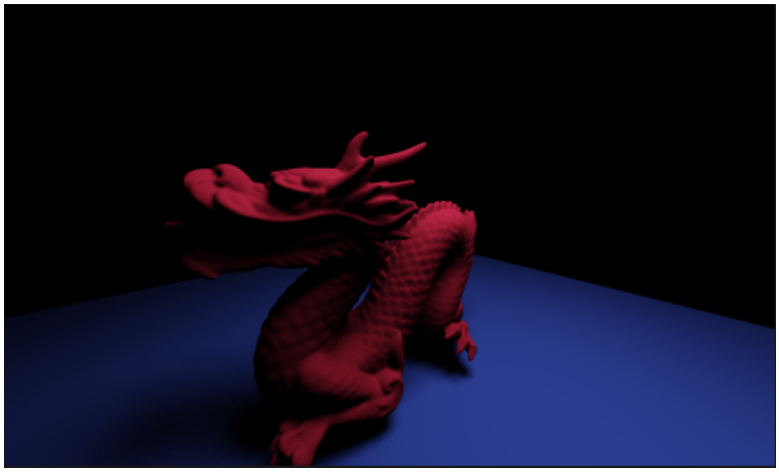
```
//面光源
std::vector<std::shared_ptr<AreaLight>> areaLights;
int nTrianglesAreaLight = 2; //面光源数（三角形数）
int vertexIndicesAreaLight[6] = { 0,1,2,3,4,5 }; //面光源顶点索引
int nVerticesAreaLight = 6; //面光源顶点数
const float yPos_AreaLight = 5.0;
Point3f P_AreaLight[6] = {
    Point3f(-1.4,yPos_AreaLight,1.4),Point3f(-1.4,yPos_AreaLight,-1.4),
       Point3f(1.4,yPos_AreaLight,1.4),
    Point3f(1.4,yPos_AreaLight,1.4),Point3f(-1.4,yPos_AreaLight,-1.4),
       Point3f(1.4,yPos_AreaLight,-1.4)
};
//面光源的变换矩阵
Transform tri_Object2World_AreaLight;
tri_Object2World_AreaLight = Translate(Vector3f(0.7f, 0.0f, -2.0f))*
   tri_Object2World_AreaLight;
Transform tri_World2Object_AreaLight = Inverse(tri_Object2World_AreaLight);
//构造三角面片集
std::shared_ptr<TriangleMesh> meshAreaLight = std::make_shared<TriangleMesh>
    (tri_Object2World_AreaLight, nTrianglesAreaLight, vertexIndicesAreaLight
       , nVerticesAreaLight,
        P_AreaLight, nullptr, nullptr, nullptr, nullptr);
std::vector<std::shared_ptr<Shape>> trisAreaLight;
//生成三角形数组
for (int i = 0; i < nTrianglesAreaLight; ++i)
    trisAreaLight.push_back(std::make_shared<Triangle>(&
       tri_Object2World_AreaLight, &tri_World2Object_AreaLight,
        false, meshAreaLight, i));
```

```
//将物体填充到基元，注意物体的基元构造函数多了面光源参数
std::shared_ptr<AreaLight> area;
......
//填充光源类物体到基元
for (int i = 0; i < nTrianglesAreaLight; ++i) {
    area = std::make_shared<DiffuseAreaLight>(tri_Object2World_AreaLight,
        Spectrum(5.f), 5, trisAreaLight[i], false);
    lights.push_back(area);
    prims.push_back(std::make_shared<GeometricPrimitive>(trisAreaLight[i],
        floorMaterial, area));
}
```

值得庆幸的是，因为面光源和点光源接口是统一的，所以我们基本不需要修改渲染积分器的 Render 函数，只需要遍历即可：

```
    for (int count = 0; count < scene.lights.size(); count++) {
        Spectrum Li = scene.lights[count]->Sample_Li(isect, pixel_sampler->
            Get2D(), &wi, &pdf_light, &vist);
        if(vist.Unoccluded(scene)){
            //计算着色
            ......
        }
    }//注意最后要把着色值再除以count
```

我们渲染得到的图像如下：



可以看到，软阴影比硬阴影好看多了。至此，我们已经拥有了光源系统。

三　镜面材质

我们本章实现一个镜子。

3 1　镜面物体

完美的镜面反射和折射（snell 定理）的公式属于初中物理的范畴。

除了反射和透射方向外，还需要计算反射或折射（透射）的的光占入射光的比例。对于物理上精确的反射或折射这些项是与方向相关的。菲涅尔方程描述了从表面反射的光量，它们是光滑表面上的麦克斯韦方程的解。

考虑到入射光线与表面法线的折射率和入射光线的角度，菲涅耳方程规定了入射光的两种不同偏振状态下材料的相应反射率。由于偏振的视觉效应在大多数环境中都是有限的，因此在 pbrt 中假设光是非偏振的，利用这种简化假设，菲涅耳反射率是平行和垂直偏振项的平方的平均值。

我们区分一下不同材料的区别：dielectrics（电介质）材料不导电，它们有实值折射率，例如玻璃、水和空气。conductors（导体）例如金属，当导体受到电磁辐射（如可见光）时，就会反射回相当一部分的光照。当然一部分光也被传输到导体内部，在那里它被迅速吸收，因为全部的吸收通常发生在材料靠近光的最外侧 0.1 m 内，因此极薄的金属薄膜就能够传输大量的光能。PBRT 中忽略了这种效应，只对导体的反射分量进行建模。Semiconductors（半导体），例如硅，也有一些特性，但 PBRT 中没有涉及。

导体和电介质都是使用的同一种菲涅尔方程，但 PBRT 还是于为电介质创建一个专门的求值函数 FrDielectric 来计算菲涅耳反射比，以便在保证折射率为实值时，从这些方程所采用的特别简单的形式中轻松得到结果。为了计算两介质界面的菲涅耳反射比，需要知道两介质的折射率，并传入 FrDielectric 函数。其中，还需要根据参数 cosThetaI 判断是物体射入了表面还是射出了表面。

FrConductor 表示光在电介质与金属之间的菲涅尔反射比（注意光一般不可能从导体内射到另一个导体内），计算公式就不列出了。

为了方便起见，PBRT 定义一个抽象的菲涅尔类，它提供了一个计算菲涅尔反射系数的接口 Fresnel，支持 FresnelConductor 和 FresnelDielectric 这两种形式的 BRDF 的实现。PBRT 其实还有一个继承自 Fresnel 的派生类 FresnelNoOp，表示百分之百反射入射光的材质（物理上难以实现）。

上面这些讲述的内容都可以直接移植到我们的系统中。

3 2　镜面反射

对于镜面反射的渲染方程可以描述为：

$$L_o(\omega_o) = \int f_r(\omega_o, \omega_i) L_i(\omega_i) |cos\theta_i| d\omega_i = F_r(\omega_r) L_i(\omega_r) \tag{三.1}$$

因此我们只需要计算出反射比即可。关于 delta 函数描述的 BRDF 可以参考 [1] 的第 8 章第 2 节，但我更喜欢直接理解镜面反射。需要注意的是镜面反射的 BRDF 表示为：

$$\frac{F_r(\omega_r)}{|cos\theta_r|} \tag{三.2}$$

该 BRDF 仅在反射方向处成立。

开始移植 SpecularReflection 类。它的实现非常简单，我们需要注意 Sample_f 函数，wi 方向其实就是计算反射的公式中，在着色空间里令 n=(0,0,1) 之后计算得到的：

```
1    −wo + 2 * Dot(wo, n) * n;
```

Sample_f 函数中，采样 Pdf 设置为 1 表示光的入射和反射方向是相关的，知道了入射方向，反射方向一定只是沿着个方向。

Pdf 函数表示给定 wo 方向，反射到 wi 的方向的概率为 0。其实我们可以理解为采样只能采样 BRDF，没法对光采样以后计算散射概率。

注意移植的时候需要把内存释放掉（当然强烈建议使用智能指针或者可以移植 PBRT 的内存管理方法）：

```
~SpecularReflection() { fresnel->~Fresnel(); }
```

因为我们没有使用 PBRT 的 MemoryArena 类，所以无法统一管理和释放内存。

镜子类定义在 Mirror.h 和 Mirror.cpp 文件中，它的实现非常简单，我们可以直接移植过去，不需要多少改动。

3 3 Whitted 光线追踪引擎的移植和测试

Whitted 光线追踪引擎之前已经说了很多遍了，这里只是告诉大家怎么移植，基本原理就不再详细讲解了。这里面的关键是 SpecularReflect 中调用的 isect.bsdf->Sample_f 函数，这个函数会根据设定的 BxDFType 来计算相应的 BSDF 值。我们等学完路径追踪和微表面材质以后再回来仔细介绍这个函数，现在我们先忽略。

我们移植一下 SamplerIntegrator::SpecularReflect 函数，函数的定义如下：

```
Spectrum SamplerIntegrator::SpecularReflect(
    const Ray &ray, const SurfaceInteraction &isect,
    const Scene &scene, Sampler &sampler, int depth) const {
    // Compute specular reflection direction _wi_ and BSDF value
    Vector3f wo = isect.wo, wi;
    float pdf;
    BxDFType type = BxDFType(BSDF_REFLECTION | BSDF_SPECULAR);
    Spectrum f = isect.bsdf->Sample_f(wo, &wi, sampler.Get2D(), &pdf, type);
    // Return contribution of specular reflection
    const Normal3f &ns = isect.shading.n;
    if (pdf > 0.f && !f.IsBlack() && AbsDot(wi, ns) != 0.f) {
        // Compute ray differential _rd_ for specular reflection
        Ray rd = isect.SpawnRay(wi);
        return f * Li(rd, scene, sampler, depth + 1) * AbsDot(wi, ns) /
               pdf;
    } else
        return Spectrum(0.f);
}
```

注意这里的 isect.bsdf 调用 Sample_f() 函数时，BxDFType 是镜面反射类型：

```
BxDFType type = BxDFType(BSDF_REFLECTION | BSDF_SPECULAR);
```

而如果表面交点不是这种类型（比如我们的龙的材质是漫反射材质），那么返回的 pdf 就是 0，此时，光线不会再去进行追踪了。虽然漫反射材质中组件也有 BSDF_REFLECTION，但 Sample_f() 函数调用的 MatchesFlags() 如果为真的前提是所有组件都要满足，也就是“镜面”、“反射”这两个条件都需要。SpecularReflect 函数只追踪镜面反射。

Li 函数在 SamplerIntegrator 类里是一个纯虚函数。我们以前都是直接在 SamplerIntegrator::Render 函数里计算的光照，现在我们定义一个派生类 WhittedIntegrator。

```
class WhittedIntegrator : public SamplerIntegrator {
    public:
        // WhittedIntegrator Public Methods
```

```
4         WhittedIntegrator(int maxDepth, std::shared_ptr<const Camera> camera
              ,
5                           std::shared_ptr<Sampler> sampler,
6                           const Bounds2i &pixelBounds)
7             : SamplerIntegrator(camera, sampler, pixelBounds), maxDepth(
                  maxDepth) {}
8         Spectrum Li(const Ray &ray, const Scene &scene,
9                     Sampler &sampler, int depth) const;
10    private:
11        // WhittedIntegrator Private Data
12        const int maxDepth;
13    };
```

我们要在 Render 函数里做一些光线 Ray 产生和结果保存的内容，Li 函数里进行光照着色。WhittedIntegrator::Li 的移植非常简单，我们暂时先注释掉下面两行就可以了：

```
1     L += isect.Le(wo);
2     L += SpecularTransmit(ray, isect, scene, sampler, arena, depth);
```

但是为了显示出地板，我们在 Li 函数前面设置第 0 次求交时如果没有击中物体就返回蓝色：

```
1     SurfaceInteraction isect;
2     if (!scene.Intersect(ray, &isect)) {
3         for (const auto &light : scene.lights) L += light->Le(ray);
4                 if (depth == 0) {
5                         Spectrum LL;
6                         LL[2] = 0.8;
7                         return LL;
8                 }
9         return L;
10    }
```

然后我们再修改一下 SamplerIntegrator::Render 函数:

```
1 for (int j = 0; j < RasterHeight; j++) {
2     for (int i = 0; i < RasterWidth; i++) {
3         int offset = (i + RasterWidth * j);
4         std::unique_ptr<Sampler> pixel_sampler = sampler->Clone(offset);
5         Point2i pixel(i, j);
6         pixel_sampler->StartPixel(pixel);
7         Spectrum colObj(0.0f);
8         do {
9             CameraSample cs;
10            cs = pixel_sampler->GetCameraSample(pixel);
11            Ray r;
12            camera->GenerateRay(cs, &r);
13            SurfaceInteraction isect;
14            //计算光照
15            colObj += Li(r, scene, *pixel_sampler, 0);
16
```

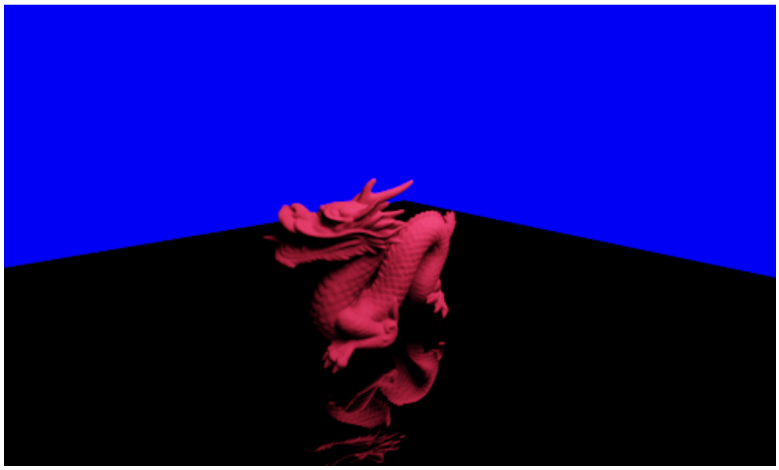
```
        } while (pixel_sampler->StartNextSample());
        colObj = colObj / (float)pixel_sampler->samplesPerPixel;
        camera->setFrameBuffer(offset, colObj, renderTimes);
    }
}
```

渲染出来的结果与以前是完全一样的。

现在我们定义并渲染一个镜面物体：

```
        Spectrum mirrorColor(1.f);
        std::shared_ptr<Texture<Spectrum>> KrMirror = std::make_shared<
            ConstantTexture<Spectrum>>(mirrorColor);
    std::shared_ptr<Material> mirrorMaterial = std::make_shared<
        MirrorMaterial>(KrMirror, bumpMap);
```

我们把该材料赋给地板，渲染得到如下场景（我将面光源设置的大了一点）：



可以看到龙底下的镜面反射。

需要提一句，我们之前在移植 Triangle::Intersect 的时候，没有给 dpdu 赋值：

```
        float determinant = duv02[0] * duv12[1] - duv02[1] * duv12[0];
        bool degenerateUV = std::abs(determinant) < 1e-8;
        if (!degenerateUV) {
                float invdet = 1 / determinant;
                dpdu = (duv12[1] * dp02 - duv02[1] * dp12) * invdet;
                dpdv = (-duv12[0] * dp02 + duv02[0] * dp12) * invdet;
        }
```

而 dpdu 会默认初始化 SurfaceInteraction 类的 shading.dpdu，再赋值给着色坐标系的 ss，因此在 Normalize 时就会因为除以 0 导致出现 NaN 非法数。所以镜面渲染的一片黑。至于为什么 matte 物体渲染的时候是正常的，这是因为这个物体在着色坐标系下只需要考虑法线方向，其他各个方向都是一样的，因此采样和计算 Pdf 时用不到坐标系的另外两个轴。我跟踪调试了很久（不断向前跟踪哪里出现了 NaN，最后定位到了 BSDF 的 Normalize 函数上），才发现以前的 Shape 中有些内容忘了去实现。

四　总结

写这本书用了不到三天时间，其中绝大部分时间都用在了 Debug 上，但哪怕一点微不足道的错误，都可能直接导致渲染不出正确结果。

我们虽然移植了 BSDF 类，但是并没有详细介绍里面的各个函数的原理，尤其是 Sample_f 函数，这个函数会根据反射特性来计算和选择反射的方向，等我们完成了路径追踪甚至微表面材质之后，再来详细分析和介绍 BSDF 类。

最后提一下调试的技巧吧。当你图像全黑的时候，你可以设置积分器的 Li 函数返回一个有颜色的值，如果正确得到单色图像，则说明是 Li 里面计算错误。之后有可能是计算 BSDF 的错误，也有可能是某个地方计算出了 NaN 值，这个时候您可以设置：

```
1        if(xxx.hasNaNs){
2        return 某种颜色;
3    }
```

不断跟踪，就能找到最终运算错误的地方。

我在设置场景的时候，不小心把地板的顶点赋值到了灯光位置上，结果计算 BSDF 每次错误，我一直以为是 BSDF 的问题，调试了很久才发现。总之，写代码不易，调试更不易，但是，哪怕经过了几个小时甚至一两天，只要我们得到了最终想要的结果，不就很愉快吗。

最后，建议大家把当前学完的所有内容再回顾和看几篇，尤其是 PBRT 的渲染流程和各个基类的关系。

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.