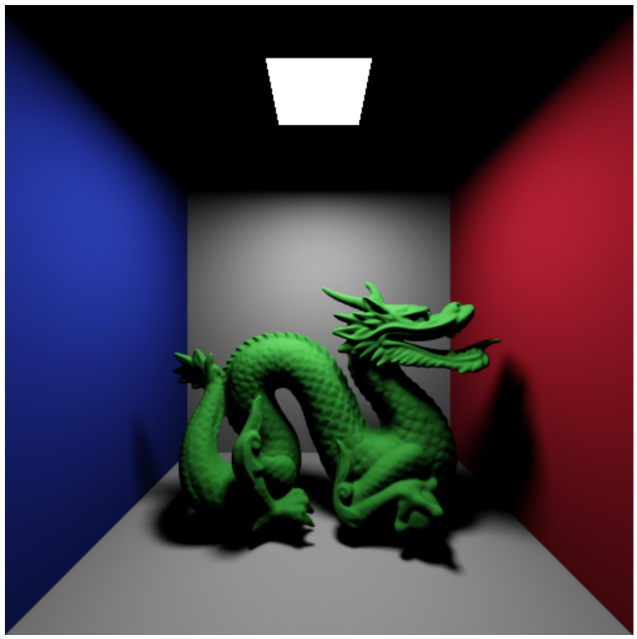
PBRT 系列 10-代码实战-一些零散和琐碎的内容补充

Dezeming Family

2021 年 3 月 11 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：学习法向量和坐标系统的反转。移植和实现完美镜面的透明材质。学习采样 BSDF，显示面光源。自己创建一个简单的环境光源。实现一个康奈尔盒。



前言

虽然我们已经将 PBRT 的主要功能移植到了我们的系统中，但很多地方其实我们都没有介绍全面，现在我们将这些地方好好的整理和归纳一下。我认为当前的整个系统的源码大家应该仔细反复看三遍以上，否则过几天你就会忘掉很多内容。

前言好像越写越少了，大概就是越来越没话可说了吧。其实在实现完加速器以后，可以说剩下的内容大家自己去研究会更好，但为了完整性（我讨厌有头无尾）我还会继续写下去。后面还要写的内容，比如最重要的路径追踪，比如我一直想写但是一直没有写的光线微分（尽管不难，但是会增加系统的复杂性），比如纹理滤波，比如 SPPM。

这本书虽然是零散和琐碎的内容，但它的重要性丝毫不低于前面的任何一本书。

本书的售价是 4 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　法向量与坐标系

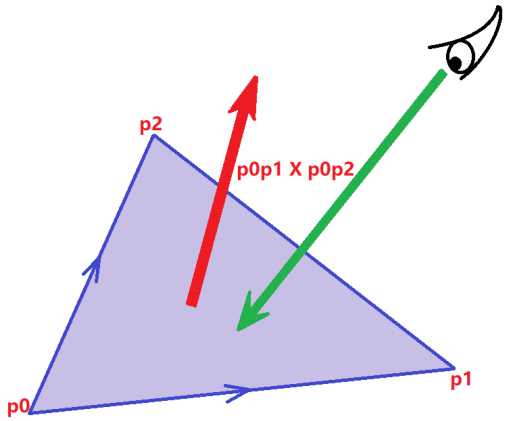
第一章讲点以前我一直故意忽略的内容。以前忽略的原因是因为真的用不到，徒增复杂性，但是现在我们好好谈谈这些内容。

1 1　法向量翻转

Shape 类里有一个成员变量：reverseOrientation。该变量表示法向量是否要进行反转。

```
1    Vector3f dp02 = p0 - p2, dp12 = p1 - p2;
2    isect->n = isect->shading.n = Normal3f(Normalize(Cross(dp02, dp12)));
```

假如我们从模型的外面看向模型（如下图），注意模型的三角形会相互遮挡，相对于人眼位置靠前的会挡住后面的。



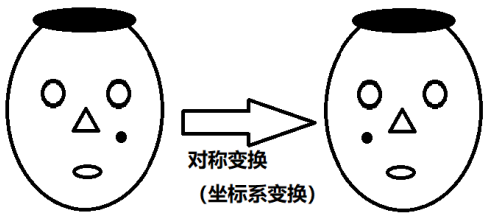
按常理来说，这时三角形的三个顶点在三角形面索引顶点数组中的顺序应该是逆时针顺序，比如 p0,p1,p2。这样计算出来的叉积就是向外的。但有的模型可能会是顺时针顺序，比如 p2,p1,p0，这样叉积以后的法向量就会朝内。

还比如我们要渲染的场景包含在一个巨大的球的内部。正常情况下我们算出的法向量是朝外的，但是我们希望不希望 Ray 与球求交后使用向外的法向量，因为光源也在球内，球将永远不会被照亮。因此我们需要设定一个法向量反转的标志。

1 2　左右手坐标系

实际上按照书本和网上的内容，关于左右手坐标系互相变换的意义并没有那么好理解。

我想到一个有趣的比喻，们可以想象，比如我们照镜子，我们的左脸有一颗痣，我们无论怎么把人旋转平移和缩放，哪怕你任意走动，你的位置进行了旋转和平移，但你的痣还是在左脸上。但是对于计算机模型，我们可以通过计算让你的痣显示在右脸上：可以把你进行对称变换，经过对称变换以后，你的痣就在右脸上了：



对称变换其实就是坐标系变了，比如本来三角形的三个点是 (1,1,1),(2,3,1),(7,2,5)，叉积得到的法向量方向为：

```
1    Point3f p0(1,1,1),p1(2,3,1),p2(7,2,5);
2    Vector3f N = Cross((p1-p0),(p2-p0));
```

得到 N 的结果是 (8,-4,-11)

我们将法向量沿着 z 轴做个对称（相当于左手坐标系变换到右手坐标系）：

```
Point3f p0(1,1,-1),p1(2,3,-1),p2(7,2,-5);
Vector3f N = Cross((p1-p0),(p2-p0));
```

得到结果为 (-8,4,-11)。但问题是我们只是让 z 轴做了个对称，因此变换后物体的法向量应该是 (8,-4,11) 才对，所以说相当于所有坐标需要取负。这就对应了为什么需要翻转：

```
if (reverseOrientation ^ transformSwapsHandedness)
        isect->n = isect->shading.n = -isect->n;
```

Shape 类里的成员变量 transformSwapsHandedness 表示是不是变换中换了坐标系。某些类型的变换会将左手坐标系更改为右手坐标系，反之亦然。有些例程需要知道源坐标系的与目标坐标系是否是不同手坐标系。特别是要确保曲面法线始终指向曲面的“外部”的例程可能需要在变换后翻转法线的方向（如果用手习惯发生变化）。判断是否被变换改变很容易：它只发生在变换的左上 3×3 子矩阵的行列式为负时，该程序即 Transform::SwapsHandedness()。

知道了原理以后，我们就可以把 transformSwapsHandedness 和 reverseOrientation 的相关内容移植到自己的系统里了（虽然一般也用不到）。

二　透明物体

我们上本书只解决了镜面物体，我们本章把透明类物体实现一下。

2 1　完美镜面透明物体的表示

我们用 τ 来表示入射能量的一部分，它被传输到由菲涅耳方程给出的输出方向。传输的功率微分为：$d\Phi_o = \tau d\Phi_i$，因为 radiance 的表示为：

$$L = \frac{d\omega}{d\omega A^{\perp}} \tag{二.1}$$

因此功率微分就可以表示为：

$$L_o cos\theta_o dA d\omega_o = \tau L_i cos\theta_i dA d\omega_i \tag{二.2}$$

把立体角展开为球坐标角表示，我们得到：

$$L_o cos\theta_o dA sin\theta_o d\theta_o d\phi_o = \tau L_i cos\theta_i dA sin\theta_i d\theta_i d\phi_i \tag{二.3}$$

Snell 定律：$\eta_i sin\theta_i = \eta_o sin\theta_o$，微分一下，得到：$\eta_i sin\theta_i d\theta_i = \eta_o sin\theta_o d\theta_i$

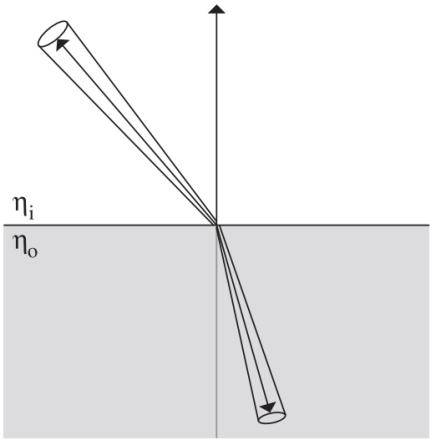
代入上式，可得：$L_o \eta_i^2 d\phi_o = \tau L_i \eta_o^2 d\phi_i$。因为在不同的两个半球，我们可以认为 $\phi_i = \phi_o + \pi$，因此 $d\phi_i = d\phi_o$。

由此我们得到镜面反射的 BTDF：

$$f_r(\omega_o, \omega_i) = \frac{\eta_o^2}{\eta_i^2}(1 - F_r(\omega_i))\frac{1}{cos\theta_i} \tag{二.4}$$

该公式仅在 Snell 定律根据法向量和折射方向计算出的入射方向 ω_i 处成立。

在具有不同折射率的介质之间的边界处，透射的辐射量按两个折射率的平方比缩放。直观地说，这可以理解为辐射的微分立体角由于传输而被压缩和扩展的结果，如下图。但是，实际计算的结果表明，这种缩放应该应用于从光源开始的光线，决不能应用于从相机开始的光线（参见双向光传输）。



通过 $1 - F_r(\omega_i)$ 项（系列 9 讲解过）我们可以直观感受到，越垂直入射，折射的光越强。

2 2　完美透明物体的移植

SpecularTransmission 类的移植比较容易。我们主要需要在意里面的几个函数。f() 函数返回的自然是 0，因为如果给定一个入射方向 wi，对于完美镜面来说，几乎不可能正好与折射方向相对应，因此返回黑色。Pdf() 函数也是如此，返回 0。Sample_f() 函数的原理都已经讲过了，需要注意的是我们提到过，实际实现中如果光线不是从光源传来的，那么就不能应用缩放。

注意 Refract 函数是求折射方向的，该函数在在 [2] 中讲过，而且网上也有大量的资料，因此就不再赘述了。这里需要注意的是，如果发生了全反射，则 Refract 会返回 false，之后 Sample_f 会返回黑色。即 SpecularTransmission 不考虑全反射的情况（FresnelSpecular 会模拟全反射、折射效果）。

鉴于 PBRT 中并没有直接对完美镜面建模的材料（glass 类包含了比较复杂的多种模型）。因此我们自己实现一种完美镜面折射的材料（大家可以自己先尝试根据 MirrorMaterial 类来实现一个完美的玻璃类，其实很简单。如果写的过程中遇到了困难，就可以看看下面我的方案）。

```
// PerfectGlassMaterial Declarations
class PerfectGlassMaterial : public Material {
    public:
    // PerfectGlassMaterial Public Methods
        PerfectGlassMaterial(const std::shared_ptr<Texture<Spectrum>> &r,
            const std::shared_ptr<Texture<Spectrum>> &t,
        const std::shared_ptr<Texture<float>> &id, const std::shared_ptr<
            Texture<float>> &bump) {
            Kr = r;
            Kt = t;
            index = id;
            bumpMap = bump;
    }
    void ComputeScatteringFunctions(SurfaceInteraction *si, TransportMode
        mode,
                                    bool allowMultipleLobes) const;
    private:
    // PerfectGlassMaterial Private Data
    std::shared_ptr<Texture<Spectrum>> Kr;
    std::shared_ptr<Texture<Spectrum>> Kt;
    std::shared_ptr<Texture<float>> index;
    std::shared_ptr<Texture<float>> bumpMap;
};
```

其中 ComputeScatteringFunctions 函数表示为：

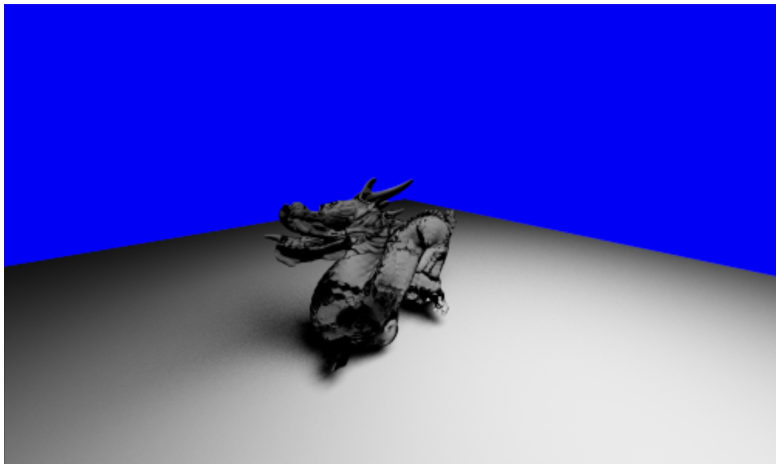
```
// PerfectGlassMaterial Method Definitions
void PerfectGlassMaterial::ComputeScatteringFunctions(SurfaceInteraction *si
    , TransportMode mode, bool allowMultipleLobes) const {
    // Perform bump mapping with _bumpMap_, if present
    //if (bumpMap) Bump(bumpMap, si);
    si->bsdf = new BSDF(*si);
    float eta = index->Evaluate(*si);
    Spectrum R = Kr->Evaluate(*si).Clamp();
    Spectrum T = Kt->Evaluate(*si).Clamp();
    if (!R.IsBlack() && !T.IsBlack())
        si->bsdf->Add(new FresnelSpecular(R, T, 1.f, eta, mode));
        //或者可以使用:
        //si->bsdf->Add(new SpecularTransmission(T, 1.f, eta, mode));
}
```

以及实现 SamplerIntegrator::SpecularTransmit 函数，该函数不需要进行多少修改。

之后我们构建该材料，并把渲染物体的材料改为该材料：

```
1        std::shared_ptr<Texture<float>> glassEta = std::make_shared<
             ConstantTexture<float>>(1.2f);
2        std::shared_ptr<Texture<Spectrum>> glassKr = std::make_shared<
             ConstantTexture<Spectrum>>(1.0f);
3        std::shared_ptr<Texture<Spectrum>> glassKt = std::make_shared<
             ConstantTexture<Spectrum>>(1.0f);
4     std::shared_ptr<Texture<float>> bumpMap = std::make_shared<
         ConstantTexture<float>>(0.0f);
5     std::shared_ptr<Material> glassMaterial = std::make_shared<
         PerfectGlassMaterial>(glassKr, glassKt, glassEta, bumpMap);
```

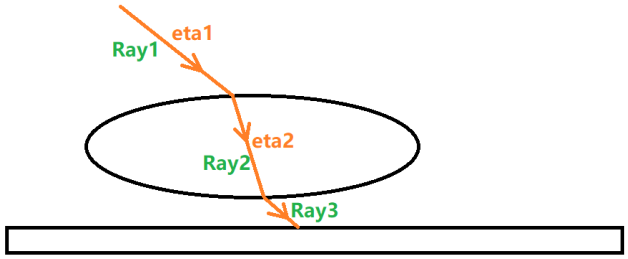
之后渲染显示结果如下：



虽然移植很容易，但是折射的过程、折射率计算方向的生成等步骤相对比较繁杂，下一节希望大家对照着源码，从 Li 函数开始，多看几遍，把流程走通。

2 3　折射的计算流程

我们以如下图示为例，我们设场景是真空的，η 值为 1.0f，玻璃的 η 值是 1.2f：



在后面的表述中，Li() 函数即 WhittedIntegrator::Li。

SamplerIntegrator::SpecularTransmit 我们简写为 SpecularTransmit。首先 Ray1 是相机发出的 Ray 的方向，Li() 函数求交后计算散射：

```
1     isect.ComputeScatteringFunctions(ray);
```

计算时生成 BSDF：

```
1     si->bsdf->Add(new SpecularTransmission(R, T, 1.f, eta, mode));
```

这时，SpecularTransmission 反射模型里的 etaA 的值是 1,0f，etaB 的值是 1.2f。

之后 Li 函数：

```
1     L += SpecularTransmit(ray, isect, scene, sampler, depth);
```

在 SpecularTransmit 函数里采样 BSDF:

```
Spectrum f = bsdf.Sample_f(wo, &wi, sampler.Get2D(), &pdf,
        BxDFType(BSDF_TRANSMISSION | BSDF_SPECULAR));
```

因为完美镜面材质只有一个 BxDF 模型,所以相当于直接调用了 SpecularTransmission 类的 Sample_f 函数。该 Sample_f 函数首先根据法向量判断是入射还是出射。对于 Ray1 到 Ray2 的过程来说，是入射。注意下面代码中的 wo 是 Ray1 的反方向，前面的书中说过，wi 和 wo 相对于交点平面都是从交点处向外的方向，而不是从外面射到交点的方向（这里的内外只是相对于交点来说的，与物体内外无关）。

```
bool entering = CosTheta(wo) > 0;
float etaI = entering ? etaA : etaB; //etaI此时为1.0
float etaT = entering ? etaB : etaA; //etaT此时为1.2
```

之后 SpecularTransmission 类的 Sample_f 函数调用 Refract 函数计算出了折射方向（而此时忽略全反射方向）。然后再计算因折射的立体角带来的能量缩放值 ft。

而后使用新生成的折射方向，得到 Ray2，然后继续做光线追踪。

三　采样 BSDF 和显示面光源

我们首先先来回顾一下材质的采样，Shape 的采样与光源的采样。对光采样等内容已经在 [4] 中早有详细的描述，所以我不介绍的特别详细，只是做一个总结。

3 1　不同函数中的采样

在 BxDF 中：

```
//f 函数表示我们已知光的入射方向wi，求从wi散射到wo方向的光分布量
//wi方向一般是通过光采样得到的方向。
f(const Vector3f &wo, const Vector3f &wi)
//Pdf 函数表示我们已知光的入射方向wi，求当出射方向为wo方向时，根据光分布，计算入射方向为wi方向的概率
//wi方向一般是通过光采样得到的方向。
Pdf(const Vector3f &wo, const Vector3f &wi)
//上述两种函数都是给定已知的wi方向来计算分布量和概率，注意对于完美镜面物体来说，我们几乎不可能除了BSDF之外的其他方式根据反射方向wo得到入射方向wi
//Sample_f是根据BSDF来进行采样。根据wo方向获得光的入射方向wi，及其概率密度。然后再调用f函数返回基于BxDF计算的光分布。
Sample_f(const Vector3f &wo, Vector3f *wi,
                             const Point2f &sample, Float *pdf,
                             BxDFType *sampledType = nullptr)
```

在 Shape 中：

```
//Sample的意义在于返回该物体上的任意一个点，并计算这个点在物体上的概率密度（即1除以面积）
Interaction Sample(const Point2f &u, Float *pdf)
//返回任意一个点在表面上的概率密度
Float Pdf(const Interaction &) const { return 1 / Area(); }
//下面的两个计算都是假设当前Shape是面光源，给定ref表示某个点，计算本面光源对该点的照明参数
//此时Shape为面光源，给一个交点，求对光采样的概率密度，返回对光采样中光源上的交点
Interaction Sample(const Interaction &ref, const Point2f &u,
                             Float *pdf)
//给定交点和入射方向，首先判断光源是否能沿着wi方向照射到本面光源上，然后计算对光采样的Pdf
Float Pdf(const Interaction &ref, const Vector3f &wi)
```

在 DiffuseAreaLight 中：

```
//Sample_Li调用的是Shape的采样函数，从Shape表面上采样一个点，得到wi方向和概率密度
Spectrum Sample_Li(const Interaction &ref, const Point2f &u, Vector3f *
    wo,
float *pdf, VisibilityTester *vis)
//Pdf_Li给定交点和入射方向，首先判断光源是否能沿着wi方向照射到本面光源上，然后计算对光采样的Pdf
```

```
    Float Pdf_Li(const Interaction &, const Vector3f &)
    //Sample_Le从面光源上找一个点，然后该点以cos权重的概率发出光，方向为wi，
        并得到采样方向的概率和采样位置的概率。
    Spectrum Sample_Le(const Point2f &u1, const Point2f &u2, float time,
    Ray *ray, Normal3f *nLight, float *pdfPos,
    float *pdfDir)
    //Pdf_Le调用的是Shape的Pdf函数，并计算采样的位置概率（1除以面积）和方向
        概率
    void Pdf_Le(const Ray &, const Normal3f &, float *pdfPos,
    float *pdfDir)
```

DiffuseAreaLight 中的 Sample_Le 和 Pdf_Le 函数主要应用于双向光传输算法，例如光子映射，我们当前只是简单了解一下。后面的系列中当用到时会仔细讲解。

3 2 采样 BSDF

之前我们采样 BxDF 都是一些独立的 BxDF，对于一个材料来说，可能需要包含多种 BxDF 反射模型，因此，BSDF 也有一个 Sample_f 函数，该函数调用存储的每个 BxDF 来采样。这里我们将从它们各自概率密度平均值的概率密度中取样：

$$p(\omega) = \frac{1}{N}\sum_{i}^{N} p_i(\omega) \tag{三.1}$$

Sample_f 函数表示如下：

```
    //选择采样哪个BxDF
    //把随机数重映射到[0,1)范围（之前使用随机数来选择采样哪个BxDF，所以不能
        重用它来继续采样，映射方法很简单，参考[1]的第14章）
    //使用所选择的BxDF来采样入射光方向
    //计算匹配的sampledType类型的全部的Pdf
    //计算采样方向的BSDF值
```

给定采样方向，此方法需要计算 BSDF 中所有相关分量的方向对 (ω_i, ω_o) 的 BSDF 值，除非采样方向来自镜面反射分量，在这种情况下，使用之前 Sample_f() 得到的值。（如果一个镜面反射组件生成了这个方向，它的 BxDF::f() 方法将返回黑色，即使我们传回它的采样返回的方向。）

虽然在计算光分布时可以只调用 BSDF::f() 方法来计算 BSDF 的值,但是可以通过直接调用 BxDF::f() 方法来更有效地计算该值，这里我们已经有了世界空间和反射坐标系中的方向。

这里提一句。以前我特别不理解为什么 f() 函数最后不需要除以 matchingComps。按理来说，如果一个材料时一堆光分布组成的，比如我使用两种 Lambertian 反射模型混合到一个 BSDF 里（虽然这么做实际没什么意义）：

```
    if (!r.IsBlack()) {
                if (sig == 0) {
                        si->bsdf->Add(new LambertianReflection(r1));
            si->bsdf->Add(new LambertianReflection(r2));
                }
```

这样渲染出来的结果，不就比以前亮了两倍么？

但其实我们在设计一种材料的时候，需要考虑多种因素，包括反射能量守恒。尤其是对于自身不发光的物体，它反射的光再多也不能超过 (1.0f,1.0f,1.0f)，例如对于完全的漫反射材料，BxDF 最高是

$(1.f, 1.f, 1.f)/\pi$，比如上面代码两种不同颜色的漫反射模型构成同一种材质的情况，r1+r2 的 RGB 任何分量都不能超过 1.0f。因此这是我们设计材料需要注意的，而不是渲染时需要注意的。换句话说，如果 f() 函数最后需要除以 matchingComps 才是见了鬼了。

3 3 显示面光源

在 Primitive 类中补充如下函数（我们之前删掉了）：

```
1    virtual const AreaLight *GetAreaLight() const = 0;
```

在除了 GeometricPrimitive 的其他派生类中实现：

```
1    const AreaLight *GetAreaLight() const { return nullptr; }
2    const Material *GetMaterial() const { return nullptr; }
```

GeometricPrimitive 里的这两个函数同样直接移植源码即可。

之后在 SurfaceInteraction 类中加入成员函数 Le，同样可以直接移植 PBRT 源码中的该函数。

我们在 WhittedIntegrator::Li 函数中不再屏蔽该代码：

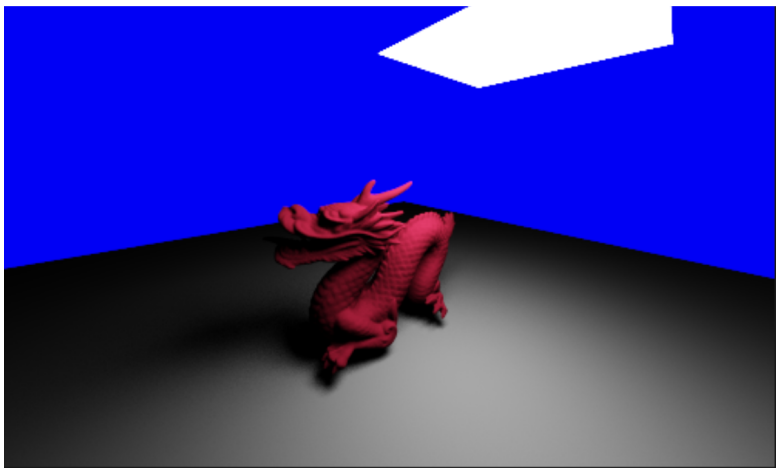
```
1    // Compute emitted light if ray hit an area light source
2    L += isect.Le(wo);
```

我们还需要为面光源的构造传入材质：

```
1    //prims是基元数组，floorMaterial是白色漫反射材质（其实最好是黑色漫反射，
         这样Ray击中光以后就可以停止追踪了）
2    prims.push_back(std::make_shared<GeometricPrimitive>(trisAreaLight[i],
         floorMaterial, area));
```

如果没有赋予面光源材质对象，在 Whitted 积分器的 Li 函数中，检测到 (!isect.bsdf)，本来是检测是否在介质边界的，现在因为没有 bsdf 信息而产生新光线去采样，也就无法显示与光源相交后光源的颜色了。

渲染结果如下：



注意如果我们是从光源的背面看向光源，则是黑的（单面光）。

四　无限环境光源

为了让图像显示得更好看，我们给渲染加个“天空盒”。

4 1　无限面光源简介

无限面光源是一种无限远的区域光源，环绕整个场景。将这种光可视化的一种方法是将光从各个方向投射到场景中的巨大球体。无限面光源的一个重要用途是环境照明，在环境中，表示照明的图像用于照亮合成对象，就像它们在该环境中一样。InfiniteAreaLight 类在 lights 文件夹下的 infinite.h 和 infinite.cpp. 中。

与其他灯光一样，InfiniteArealLight 也采用变换矩阵。这里变换矩阵的用途是确定图像贴图的方向。然后使用球坐标从球面上的方向映射到 (θ, ϕ) 方向，然后从那里映射到 (u, v) 纹理坐标。

InfiniteArealLight 的构造器加载图像并创建 MIPMap 来存储它。加载数据的非常简单。构造函数中还会初始化无限区域光的采样 Pdf，与无限区域光的蒙特卡罗采样有关，这个我们会在后面的系列，关于双向光传输算法中介绍。

该类中的成员变量 worldCenter 表示无限面光源的中心在世界坐标的何处，worldRadius 表示无限面光源的半径大小。Lmap 表示 MIPMap，Le() 方法输入 Ray，调用 Lmap->Lookup 查询无限面光源的纹理。

关于无限面光源和 MIPMap 的实现我觉得都能写一个系列了（因此我放到了系列 13《无限面光源》），所以我们当前先自己实现一个类，直接根据该类生成无限面光源。

4 2　简易无限环境光的创建和测试

我们还是假设无限环境光是一个球体，这个球体的中心一般在世界坐标的原点（其实不在世界坐标的中心也很简单，我们对求交后的点做一个偏移再映射到以原点为中心的球上即可，只是放在世界坐标的中心全景图显示会比较好看，或者放在世界坐标下的相机原点位置）。

我们先定义自己的天空盒类：

```
1   // SkyBoxLight Declarations
2   class SkyBoxLight : public Light {
3     public:
4       // SkyBoxLight Public Methods
5         SkyBoxLight(const Transform &LightToWorld,const Point3f&
              worldCenter, float worldRadius, int nSamples)
6           : Light((int)LightFlags::Infinite, LightToWorld, nSamples),
7             worldCenter(worldCenter),
8             worldRadius(worldRadius){ }
9       void Preprocess(const Scene &scene) {}
10      Spectrum Power() const { return Spectrum(0.f); }
11      Spectrum Le(const Ray &ray) const;
12      Spectrum Sample_Li(const Interaction &ref, const Point2f &u,
            Vector3f *wi,
13                         float *pdf, VisibilityTester *vis) const { return
                               Spectrum(0.f); }
14      float Pdf_Li(const Interaction &, const Vector3f &) const { return
            0.f; }
15      Spectrum Sample_Le(const Point2f &u1, const Point2f &u2, float time,
16                         Ray *ray, Normal3f *nLight, float *pdfPos,
17                         float *pdfDir) const { return Spectrum(0.f); }
```

```
18          void Pdf_Le(const Ray &, const Normal3f &, float *pdfPos, float *
                pdfDir) const {}
19        private:
20          // SkyBoxLight Private Data
21          Point3f worldCenter;
22          float worldRadius;
23      };
```

然后我们先为了测试，实现一个简单的 Le 接口：

```
1       Spectrum SkyBoxLight::Le(const Ray &ray) const {
2           Vector3f oc = ray.o - worldCenter;
3           float a = Dot(ray.d, ray.d);
4           float b = 2.0 * Dot(oc, ray.d);
5           float c = Dot(oc, oc) - worldRadius * worldRadius;
6           float discriminant = b*b - 4 * a*c;
7           float t;
8           if (discriminant < 0)
9               return 0.f;
10          t = (-b + sqrt(discriminant)) / (2.0 * a);
11          Point3f hitPos = ray.o + t * ray.d;
12          //偏移到使球心在坐标原点
13          Vector3f hitPos_temp = hitPos - worldCenter;
14          Spectrum Col;
15          Col[0] = (hitPos_temp.x + worldRadius) / (2.f * worldRadius);
16          Col[1] = (hitPos_temp.y + worldRadius) / (2.f * worldRadius);
17          Col[2] = (hitPos_temp.z + worldRadius) / (2.f * worldRadius);
18          return Col;
19      }
```

上面的函数首先先计算 Ray 与球的求交 [2]，然后找远交点（因为我们的相机在球内部，近交点在相机背面），之后为了先显示个结果，我们将交点坐标值映射到 [0,1] 之间，得到颜色 Col。

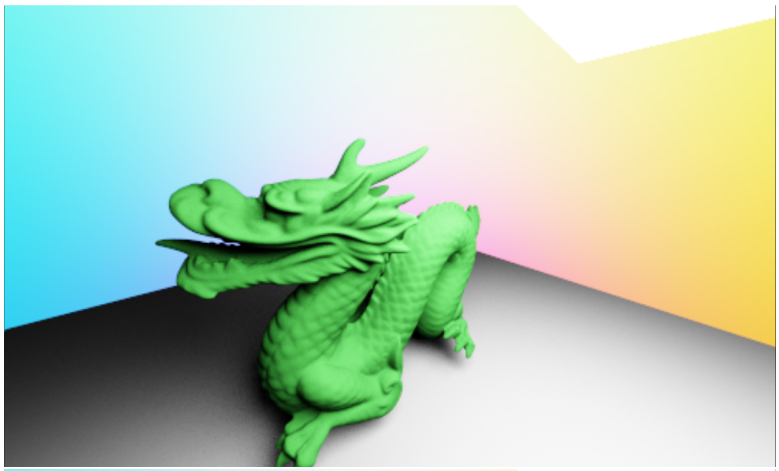
场景构建中，为了好看我给龙换了一个颜色 (0.2,0.7,0.2)，然后构造无限环境光源：

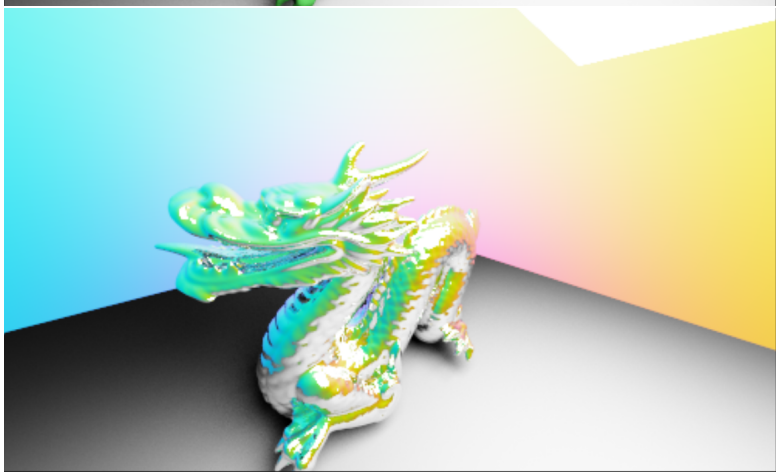
```
1           Transform InfinityLightToWorld;
2           Point3f InfinityLightCenter(0.f,0.f,0.f);
3           float InfinityLightRadius = 4.0f;
4           std::shared_ptr<Light> infinityLight =
5                   std::make_shared<SkyBoxLight>(InfinityLightToWorld,
                        InfinityLightCenter, InfinityLightRadius, 1);
6           lights.push_back(infinityLight);
```

之后在 WhittedIntegrator::Li 函数中的：

```
1       if (!scene.Intersect(ray, &isect)) {
2           for (const auto &light : scene.lights) L += light->Le(ray);
3           return L;
4       }
```

当 Ray 与场景物体没有相交时，就会通过 Le 函数得到环境光的值。渲染结果如下，第一张是漫反射材质，第二张是镜面材质。





4 3 载入图像的环境光源

我们从 [5] 中得到图像加载函数文件：stb_image.h，该文件对于可加载的图像类型非常的齐全。在 SkyBoxLight 类中加入：

```
    //成员函数
    ~SkyBoxLight() {
        if (data) free(data);
    }
        bool loadImage(char* imageFile);
    Spectrum getLightValue(float u, float v) const;
    //成员变量
        int imageWidth, imageHeight, nrComponents;
        float *data;
```

构造函数体内实现：

```
    imageWidth = 0;
    imageHeight = 0;
    nrComponents = 0;
    data = nullptr;
    //file是构造函数的传入参数，表示图像文件在磁盘中的路径
    loadImage(file);
```

析构函数：

```
~SkyBoxLight() {
    if (data) free(data);
```

```
}
```

将单位球面上的三维坐标转为球面坐标，并映射到全景图的函数如下。全景图的 width 表示 ϕ 角，坐标从 0 到 2π，height 表示 θ 角，坐标从 0 到 π。get_sphere_uv[3] 得到的是归一化纹理。

```
    void get_sphere_uv(const Vector3f&p, float &u, float &v) {
        float phi = atan2(p.z, p.x); //[-pi,pi]
        if (phi < 0.f)phi += 2 * Pi;
        float theta = Pi - acos(p.y);
        u = phi * Inv2Pi;
        v = theta * InvPi;
    }
//输入为归一化纹理
Spectrum SkyBoxLight::getLightValue(float u, float v) const{
    int w = u * imageWidth, h = v * imageHeight;
    int offset = (w + h*imageWidth) * nrComponents;
    Spectrum Lv;
    Lv[0] = data[offset + 0];
    Lv[1] = data[offset + 1];
    Lv[2] = data[offset + 2];
    return Lv;
}
```

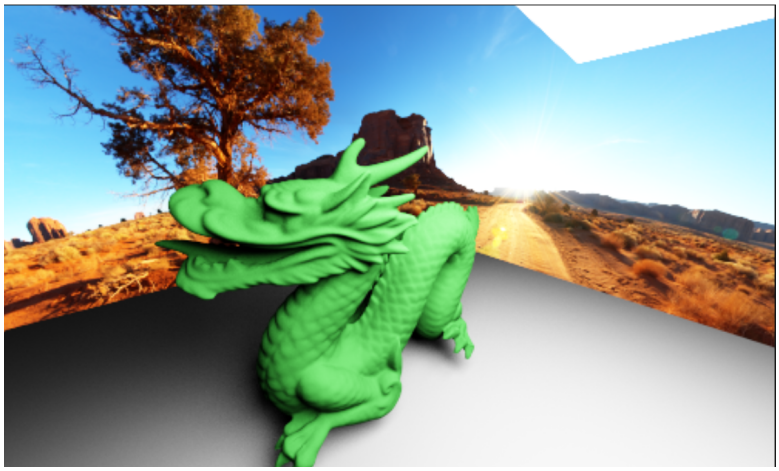
加载图像数据如下。HDR 全景文件我们可以从 [6] 中下载。全景图是映射到球面坐标的图像，其中横轴表示从 0 到 2π 的 ϕ 值，纵轴表示从 0 到 π 的 θ 值。

```
bool SkyBoxLight::loadImage(char* imageFile) {
    //图像纵坐标翻转，使图像(0,0)点位于左下角。
    stbi_set_flip_vertically_on_load(true);
    data = stbi_loadf(imageFile, &imageWidth, &imageHeight, &nrComponents,
        0);
    if (data)return true;
    else return false;
}
```

SkyBoxLight::Le 函数我们也要改一下：

```
Spectrum Col;
if (data) {
    Col = getLightValue(u, v);
}
else {
    Col[0] = (hitPos_temp.x + worldRadius) / (2.f * worldRadius);
    Col[1] = (hitPos_temp.y + worldRadius) / (2.f * worldRadius);
    Col[2] = (hitPos_temp.z + worldRadius) / (2.f * worldRadius);
}
return Col;
```

编译后运行结果如下：



4 4　对无限环境光采样

我们当前只是显示出了无限面光源。但是其实我们并没有用该面光源来进行照明，因为 SkyBoxLight 的 Sample_Li 函数只是返回了黑色，也没有给入射方向和概率密度。

我们暂时还没有实现基于无限面光源的重要性采样（即亮的地方多一些采样点），因此我们就按照均匀采样就好了。但是我们控制一下 getLightValue 的返回值大小（我们还没有 HDR 到 LDR 的映射函数），否则图像太亮了：

```
1    SkyBoxLight::getLightValue:
2    //x表示0,1,2
3    Lv[x] = data[offset + x] / 5.f;
```

Sample_Li 函数较为简单，注意我们的 LightToWorld 其实就是没有变换的单位矩阵。

```
1  Spectrum SkyBoxLight::Sample_Li(const Interaction &ref, const Point2f &u,
       Vector3f *wi,
2          float *pdf, VisibilityTester *vis) const {
3          float theta = u.y * Pi, phi = u.x * 2 * Pi;
4          float cosTheta = std::cos(theta), sinTheta = std::sin(theta);
5          float sinPhi = std::sin(phi), cosPhi = std::cos(phi);
6          *wi = LightToWorld(Vector3f(sinTheta * cosPhi, sinTheta * sinPhi,
               cosTheta));
7          *pdf = 1.f / (4 * Pi);
8          *vis = VisibilityTester(ref, Interaction(ref.p + *wi * (2 *
               worldRadius), ref.time));
9          return getLightValue(u.x, u.y);
10 }
```

渲染得到如下效果（因为没有光源的重要性采样，所以渲染了 1000 帧 (随机数使用的时钟随机数发生器，可以源源不断的产生随机数)）：

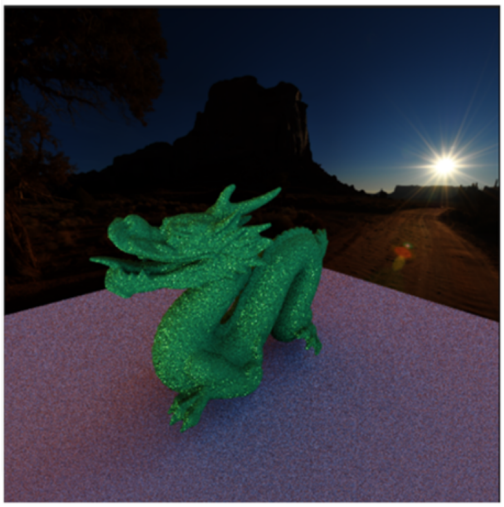
所以说，如果想要快速收敛的话，还是得对无限面光源进行重要性采样。比如上图中的阳光的位置应该多采样一些点。

4 5 HDR 转 LDR

我们的光源经常具有高动态范围，例如可能 Radiance 达到几十甚至上百，这时，渲染出来的图像颜色值就很可能几十甚至上百。而我们一般是将 [0,1] 之间的值映射到 [0,255]，因此高动态范围的数据也必须压缩到 0 到 1。

```
1    Spectrum HDRtoLDR(Spectrum& col, float exposure) {
2        float invExposure = 1.0 / (1.0 - exposure);
3        float temp_c[3];
4        temp_c[0] = 1.0 - expf(-col[0] * invExposure);
5        temp_c[1] = 1.0 - expf(-col[1] * invExposure);
6        temp_c[2] = 1.0 - expf(-col[2] * invExposure);
7        return Spectrum::FromRGB(temp_c);
8    }
```

如此做，exposure 越高，整体亮度就越高，反之亦然。下图分别是同样的渲染结果采用不同的 exposure 值得到的显示效果：



五　康奈尔盒和本书结语

本章我们实现一个康奈尔盒。

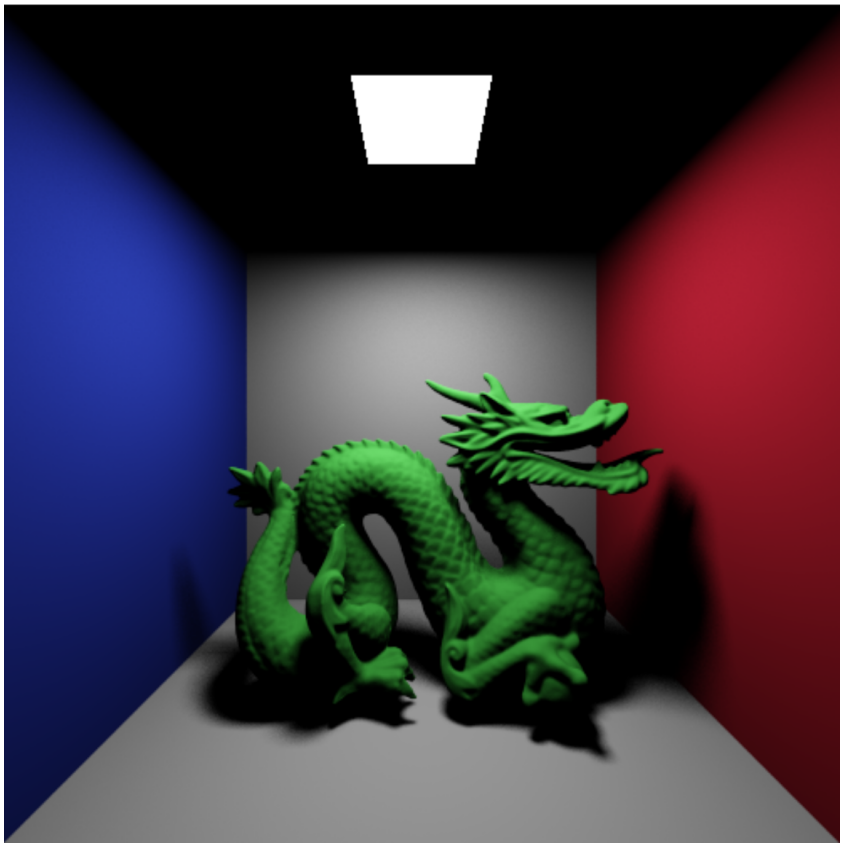
康奈尔盒的边为 5，该盒的 BoundingBox 为 (0,0,0) 和 (5,5,5)。

面光源位于包围盒的顶部靠下 0.01 处，为正方形，中心在天花板的中心。

摄像机原点位于 (2.5,2.5,6.0)，看向 (2.5,2.5,0.0) 处。

```
Point3f P_Floor[nVerticesFloor] = {
    //底座
    Point3f(0.f,0.f,length_Floor),Point3f(length_Floor,0.f,length_Floor),
        Point3f(0.f,0.f,0.f),
    Point3f(length_Floor,0.f,length_Floor),Point3f(length_Floor,0.f,0.f),
        Point3f(0.f,0.f,0.f),
    //天花板
    Point3f(0.f,length_Floor,length_Floor),Point3f(0.f,length_Floor,0.f),
        Point3f(length_Floor,length_Floor,length_Floor),
    Point3f(length_Floor,length_Floor,length_Floor),Point3f(0.f,length_Floor
        ,0.f),Point3f(length_Floor,length_Floor,0.f),
    //后墙
    Point3f(0.f,0.f,0.f),Point3f(length_Floor,0.f,0.f), Point3f(length_Floor
        ,length_Floor,0.f),
    Point3f(0.f,0.f,0.f), Point3f(length_Floor,length_Floor,0.f),Point3f(0.f
        ,length_Floor,0.f),
    //右墙
    Point3f(0.f,0.f,0.f),Point3f(0.f,length_Floor,length_Floor), Point3f(0.f
        ,0.f,length_Floor),
    Point3f(0.f,0.f,0.f), Point3f(0.f,length_Floor,0.f),Point3f(0.f,
        length_Floor,length_Floor),
    //左墙
    Point3f(length_Floor,0.f,0.f),Point3f(length_Floor,length_Floor,
        length_Floor), Point3f(length_Floor,0.f,length_Floor),
    Point3f(length_Floor,0.f,0.f), Point3f(length_Floor,length_Floor,0.f),
        Point3f(length_Floor,length_Floor,length_Floor)
};
```

然后我们将要渲染的物体移动到场景中央。最后得到渲染图：



本书写于 3 月 11 日，完稿于 3 月 14 日，用时不到 4 天。本书的意义重在实践，大家一定要多修改多练习，将引擎的构建和使用流程弄熟。如果有兴趣的话，可以自己研究并实现一种材质，也可以移植真实感相机等，在完善系统的同时让自己对系统更加熟悉。

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.

[5] https://github.com/nothings/stb/blob/master/

[6] http://www.hdrlabs.com/sibl/archive.html