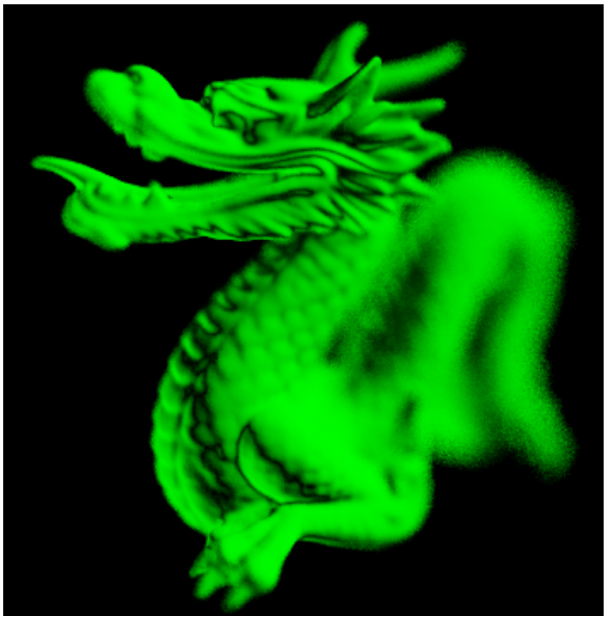
PBRT 系列 5-相机系统

Dezeming Family

2021 年 2 月 28 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 的相机系统，包括透视投影相机和正交投影相机。学习薄透镜模型和景深。在自己的引擎上移植 PBRT 相机并渲染出如下结果：



本文于 2022 年 7 月 6 日进行再版，并提供了源码。注意源码中的图形 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

上一本关于颜色和光谱的书读者学着应该是非常轻松愉快的，哪怕可能实际移植的时候遇到了一点困难，但解决完以后，自己的能力就又会提升一步！

其实，本来这本书是要写光线微分的（但考虑到光线微分主要是用来进行纹理滤波，我们的物体还没有纹理），考虑到相机似乎更简单，而且定义了相机以后，我们就可以让相机在场景中任意设置，渲染出不同角度下的物体了，岂不是更有趣！因此，本书我们就研究 PBRT 的相机系统，全景相机简单而且暂时用不到，我们不学。真实感相机我们暂时用不到而且过于复杂，我们也不学，我们只学投射相机类，正交投影相机和透视投影相机两种针孔相机模型。

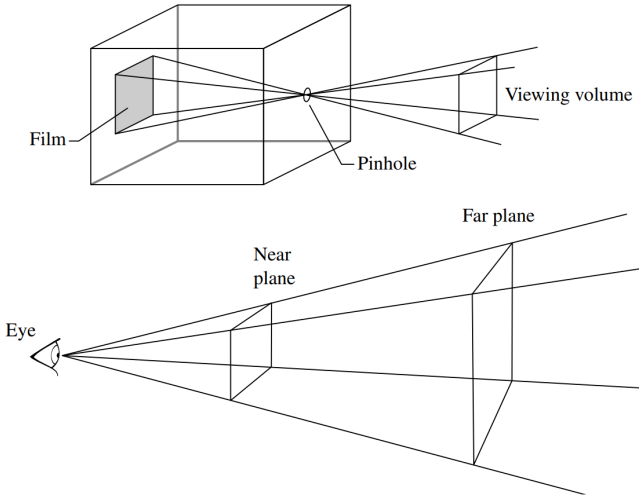
希望读者在学习本书之前，已经知道了什么是透视投影和正交投影变换，如果没有，可以参考 DezemingFamily 的《三维视角投影变换与代码实战》，或者虎书 [5] 的第 6 和第 7 章。但是 PBRT 的相机系统透视变换与一般的光栅器有很大区别!! 因此本书并没有做很多解释，大家可以在 DezemingFamily 的《三维视角投影变换与代码实战》中找到对于 PBRT 相机变换的详细的全面的解析。

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 相机类描述

针孔相机是最简单的一类相机：



上图中的上面表示的是针孔相机模型，下图是计算机图形学里常用的针孔相机表示。但是这种相机与我们真实的相机有很大不同，比如真实的相机存在离焦模糊，而针孔相机在所有区域都非常清晰。另外真实相机中，光通过镜头系统以后的分布也不同于针孔相机。但真实相机的离焦模糊现象我们是可以模拟出来的，而且真实感相机实在是有些复杂，而且一般用不到，因此我们只学习针孔相机。学会并移植好针孔相机以后，其他的相机种类并不复杂。

注意，对于双向光传输算法我们还需要定义额外的相机，等以后我们学习该算法时我们再定义，现在并不着急。

1 1　相机基类

相机基类 Camera 定义在 core 文件夹下的 camera.h 和 camera.cpp 文件里，该文件还定义了其派生类 ProjectiveCamera，在 cameras 文件夹下定义了 Camera 和 ProjectiveCamera 的各个派生类。

有点麻烦的是，相机类的构造函数的第一个输入参数是 AnimatedTransform，也就是说相机自己是可以在场地里运动的。不过我们不需要管它怎么运动的，等后面我们自己移植的时候，就直接让它静止吧，也就是使用 Transform 类。下面是 Camera 的成员变量：

```
1    //从相机坐标到世界坐标的变换
2    AnimatedTransform CameraToWorld;
3    //快门开启和关闭时间，为了处理运动模糊
4    const Float shutterOpen, shutterClose;
5    //film类表示胶片，用来存储渲染的结果
6    Film *film;
7    //参与介质（比如相机正好待在烟雾里）
8    const Medium *medium;
```

我们本书的内容只需要关注 CameraToWorld 变换，后面再讲。

相机最重要的函数是 Camera::GenerateRay()，它的第一个输入参数是一个采样结构：

```
1    struct CameraSample {
2        Point2f pFilm; //当前Ray采样到的值显示在胶片的哪个位置。
3        Point2f pLens; //当前Ray通过镜头的位置
4        Float time; //当前Ray产生的时间
5    };
```

GenerateRayDifferential() 是与光线微分相关的，但是我们暂时不去学习光线微分，所以我们现在不用管它。

摄影机放置在场景中某个世界空间点上，具有特定的观察方向和方向。此相机定义了一个新的坐标系，其原点位于相机的位置。该坐标系的 z 轴映射到观察方向，y 轴映射到向上方向。这是一个方便的空间，用于得到哪些对象对相机可能可见。例如，如果对象的相机空间边界框完全位于 z=0 平面的后面（并且相机的视野不大于 180 度），则相机将看不到该对象。

1 2 Film 胶片类

虽然我不打算使用和移植 Film 类，但是还是需要讲讲这个类是干嘛的。该类定义在 core 文件夹下的 film.h 和 film.cpp 文件中。

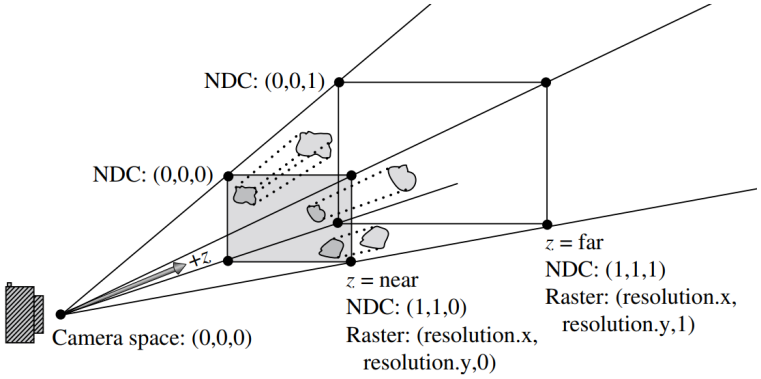
我们已经知道，Film 存储最后渲染的结果，可以简单理解为，该类保存了图像的各个像素值。PBRT 支持多线程渲染，这并不是基于 GPU 的多线程，而是基于 CPU 多核渲染的，将整张图像分成一个个小块（tile）。当小块渲染完以后，将结果融合（merged to）到图像存储区（简单来说就是拷贝过去）。当渲染完毕后，将图像保存。所以，该类里面需要什么函数，以及功能如何大家应该都明白了。有兴趣可以看 PBRT 书 [1] 的第七章，学习一下该类的具体细节，我应该以后也不会再详细讲这个类了，顶多就是遇到的时候提一下（说实话我并没有研究过该类，因为我喜欢使用 GUI 界面来显示图像，CPU 并行加速的话也倾向于直接使用简单的 OpenMP，还是那句话，人生苦短，及时行乐）。

二　投射类相机

本章我们介绍 PBRT 中派生自投射相机的两个子类：正交投影相机和透视投影相机。

2 1　投射相机基类

投射相机经常用下面的示意图（下图是透视投影相机）表示：



NDC 是标准设备坐标，在裁剪体内会被变换到 (0,0,0) 到 (1,1,1) 的包围盒（NDC，标准化设备坐标）里，然后再映射到光栅坐标（Raster）上。对于投射相机来说，它的输入参数相比相机基类 Camera 多了 Transform &CameraToScreen，即变换到屏幕坐标的变换，对于投射相机还有 RasterToScreen，RasterToCamera 等变换，就是从光栅坐标到屏幕再到相机坐标之间的变换，我们先简单介绍一下各个空间。

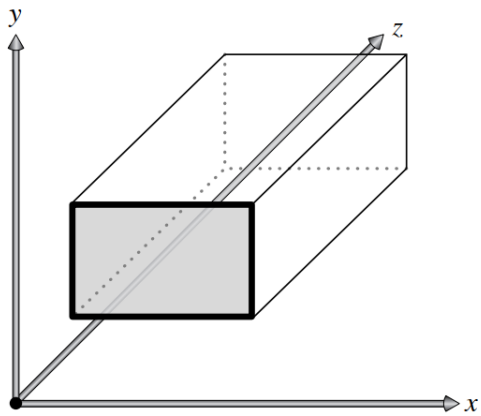
屏幕空间：屏幕空间在胶片（film）平面上定义。相机将相机空间中的对象投影到胶片平面上；屏幕窗口内的部分在生成的图像中可见。屏幕空间中的深度 z 值范围为 0 到 1，分别对应于近剪裁平面和远剪裁平面上的点。请注意，尽管这称为“屏幕”空间，但它仍然是一个三维坐标系，因为 z 值是有意义的。

屏幕空间值转 NDC 坐标值非常简单，z 值不需要改变，x 和 y 值分别除以其分辨率就好了。

光栅空间的大小可以理解为最后我们渲染出的结果的分辨率（不经过图像缩放等后处理）。也就是说，屏幕空间和 NDC 空间都是连续的空间，这里面定义的内容都是连续的，而显示器分辨率有限，所以光栅空间是离散的。

2 2　正交投影相机

正交相机最简单了，平行投影，不会产生近大远小的效果。



该相机定义在 cameras 文件夹下的 orthographic.h 和 orthographic.cpp 文件里。

当我们要渲染 film 上某个点的值时，我们将该点通过 RasterToCamera 变换到相机空间，生成 Ray。再使用 CameraToWorld 将相机空间的 Ray 变换到世界空间，然后与世界空间的物体求交计算着色。

投影相机传入参数中有 CameraToScreen，即从相机空间变换到屏幕空间。之后其他的变换都会被计算出来（见 camera.h）：

```
1    ScreenToRaster =
```

```
        Scale(film->fullResolution.x, film->fullResolution.y, 1) *
        Scale(1 / (screenWindow.pMax.x - screenWindow.pMin.x),
              1 / (screenWindow.pMin.y - screenWindow.pMax.y), 1) *
        Translate(Vector3f(-screenWindow.pMin.x, -screenWindow.pMax.y, 0));
    RasterToScreen = Inverse(ScreenToRaster);
    RasterToCamera = Inverse(CameraToScreen) * RasterToScreen;
```

正交投影相机中 screenWindow 的坐标范围是可以从 CreateOrthographicCamera 函数中得到，screenWindow 的坐标长宽比都保留着，且坐标界限的绝对值大于等于 1。CameraToScreen 的计算是：Orthographic(0, 1)（您可以在 orthographic.h 文件里看到），即让可视范围在 0 到 1 之间（我们完全可以写成 Orthographic(1,100)，其实无所谓，我们用这个函数的目的是为了从光栅空间得到某像素然后逆变换回去，不需要管它的可视范围）。

```
    //CreateOrthographicCamera函数
    Float frame = params.FindOneFloat(
        "frameaspectratio",
        Float(film->fullResolution.x) / Float(film->fullResolution.y));
    Bounds2f screen;
    if (frame > 1.f) {
        screen.pMin.x = -frame;
        screen.pMax.x = frame;
        screen.pMin.y = -1.f;
        screen.pMax.y = 1.f;
    } else {
        screen.pMin.x = -1.f;
        screen.pMax.x = 1.f;
        screen.pMin.y = -1.f / frame;
        screen.pMax.y = 1.f / frame;
    }
    //Transform.cpp
    Transform Orthographic(Float zNear, Float zFar) {
        return Scale(1, 1, 1 / (zFar - zNear)) * Translate(Vector3f(0, 0, -
            zNear));
    }
```

产生 Ray 的函数表示如下：

```
    Float OrthographicCamera::GenerateRay(const CameraSample &sample, Ray *
        ray) const {
        Point3f pFilm = Point3f(sample.pFilm.x, sample.pFilm.y, 0);
        Point3f pCamera = RasterToCamera(pFilm);
        //正交投影相机的Ray方向在相机空间都是朝向z轴正方向的
        *ray = Ray(pCamera, Vector3f(0, 0, 1));
        *ray = CameraToWorld(*ray);
        return 1;
    }
```

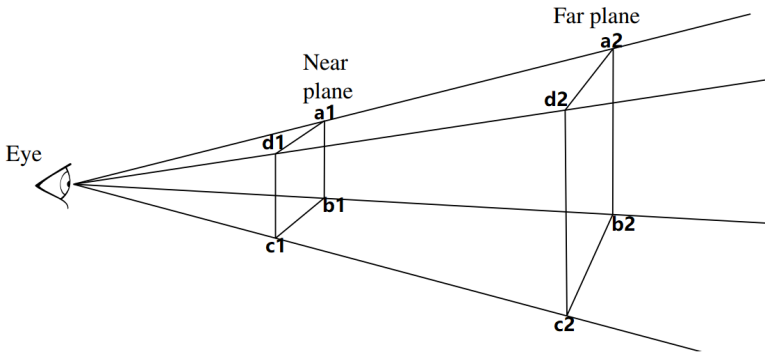
即相机空间中，相机原点在 (0,0,0) 的位置，屏幕空间大小取决于可视范围。比如当相机空间中 x 轴的可视范围为 [-100,100]，光栅空间在 x 方向的分辨率是 400，则-100 经过 ScreenToRaster 变换后就变为了 0，100 经过 ScreenToRaster 变换后就变为了 400（变换到标准坐标空间 NDC）。

2 3　透视投影相机

有了前面正交投影相机的铺垫，透视投影相机就很简单了。

透视相机定义在 cameras 文件夹下的 perspective.h 和 perspective.cpp 文件里。

构造函数传入的参数中，CameraToScreen 函数是 Perspective(fov, 1e-2f, 1000.f)。因为大家已经学习过透视投影，所以这里我只说一下它的功能，如下图：



CameraToScreen 会把近平面上的点，也就是上图中 a1,b1,c1,d1 构成的面上的点的 z 值都变为 0，远平面上的点经过变换会变为 1。处在近平面和远平面构成的视锥体内的点经过 CameraToScreen 变换以后会被压缩到 screen 空间的范围内（该范围不是严格的 $[-1, 1]$）。所以透视相机和正交相机的 CameraToScreen 函数对可见空间内的点变换后的坐标范围是不同的：z 轴都是压缩到了 $[0, 1]$，但是正交相机并没有压缩 x 和 y，而透视相机把 x 和 y 压缩到了 screen 之间，大家尤其需要注意这一点，关于具体的压缩过程和范围，大家可以从《三维视角投影变换与代码实战》中找到详细解释，该书已发布。

产生 Ray 的函数表示如下：

```
1  Float PerspectiveCamera::GenerateRay(const CameraSample &sample,
2                                       Ray *ray) const {
3      Point3f pFilm = Point3f(sample.pFilm.x, sample.pFilm.y, 0);
4      Point3f pCamera = RasterToCamera(pFilm);
5      *ray = Ray(Point3f(0, 0, 0), Normalize(Vector3f(pCamera)));
6      *ray = CameraToWorld(*ray);
7      return 1;
8  }
```

相机位置在相机坐标系的 (0,0,0) 点，Ray 的方向是相机原点到近平面上的一个点连成的向量。

三　相机测试

本章我们先把相机移植和测试成功，然后下一章再学习和实现景深。

在 PBRT 中，相机 Ray 与物体求交的思路是这样的：首先相机先在相机空间产生光线 Ray，然后变换 Ray 到世界坐标空间里；物体从物体空间变换到世界坐标系下；之后两者再互相求交。

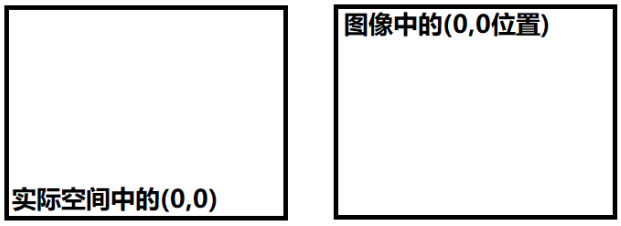
3 1　相机移植

创建 Camera.h 和 Camera.cpp 文件。Camera 和 ProjectiveCamera 类移植的时候，需要将 AnimatedTransform 参数改为 Transform，去掉 shutterOpen，shutterClose，film 和 medium 参数（我们只保留 Transform 这个参数）。成员函数只保留 GenerateRay，成员变量只保留 CameraToWorld。

CameraSample 结构只去掉 time 这个变量就好了。

ProjectiveCamera 的构造函数中，float lensr 和 float focald 两个参数都保留着就好了，因为就是两个 float 类型的变量，下一章我们学完景深之后就能用到。

```
1    //下面这一行
2    Scale(film->fullResolution.x, film->fullResolution.y, 1)
3    //film->fullResolution改为你要渲染出的图像的分辨率。
4    //比如你要生成500*300的图像，就改为：
5    Scale(500, 300, 1)
6    //下面这一行
7    1 / (screenWindow.pMin.y - screenWindow.pMax.y), 1)
8    //与上面不同，是用的pMin.y-pMax.y，这是因为如下图，图像的上下和实际空间
         中的上下是相反的。
9    //自己编程的时候一定要注意这一点
```



然后创建 orthographic 和 perspective 两个类的相关文件进行移植。

dxCamera 和 dyCamera 及其相关操作我们暂时都去掉。我们只保留里面的 GenerateRay 函数，直接用前面我们附带的代码就好了，先删除该函数里与透镜和聚焦有关的内容。

CreateOrthographicCamera 的移植如下：

```
1    OrthographicCamera *CreateOrthographicCamera(const Transform &cam2world)
         {
2        //取决你渲染图像的分辨率
3        float frame = (float)500 / (float)300;
4        Bounds2f screen;
5        if (frame > 1.f) {
6            screen.pMin.x = -frame;
7            screen.pMax.x = frame;
8            screen.pMin.y = -1.f;
9            screen.pMax.y = 1.f;
10       }
11       else {
```

```
            screen.pMin.x = -1.f;
            screen.pMax.x = 1.f;
            screen.pMin.y = -1.f / frame;
            screen.pMax.y = 1.f / frame;
        }
        //这是相机空间中实际能看到的范围，我设置的大一些。
        float ScreenScale = 2.0f;
        {
            screen.pMin.x *= ScreenScale;
            screen.pMax.x *= ScreenScale;
            screen.pMin.y *= ScreenScale;
            screen.pMax.y *= ScreenScale;
        }

        float lensradius = 0.0f; //保留变量
        float focaldistance = 0.0f; //保留变量
        return new OrthographicCamera(cam2world, screen
            ,lensradius, focaldistance);
    }
```

PerspectiveCamera 不需要保留任何私有成员变量。

```
    PerspectiveCamera::PerspectiveCamera(const Transform &CameraToWorld,
        const Bounds2f &screenWindow, float lensRadius, float focalDistance,
            float fov)
        : ProjectiveCamera(CameraToWorld, Perspective(fov, 1e-2f, 1000.f),
            screenWindow, lensRadius, focalDistance) {

        // 计算相机在相机空间的z=0处能看到的范围
        //这是PBRT用来插值和估计用的，但是因为可能会很有用，所以我们保留这几
            行。
        Point2i res = Point2i(500, 300);
        Point3f pMin = RasterToCamera(Point3f(0, 0, 0));
        Point3f pMax = RasterToCamera(Point3f(res.x, res.y, 0));
        pMin /= pMin.z;
        pMax /= pMax.z;
    }
```

CreatePerspectiveCamera 函数改成下面这个样子（500*300 分辨率）：

```
PerspectiveCamera *CreatePerspectiveCamera(const Transform &cam2world) {
    float lensradius = 0.0f;
    float focaldistance = 0.0f;
    float frame = (float)500 / (float)300;
    Bounds2f screen;
    if (frame > 1.f) {
        screen.pMin.x = -frame;
        screen.pMax.x = frame;
        screen.pMin.y = -1.f;
```

```
        screen.pMax.y = 1.f;
    }
    else {
        screen.pMin.x = -1.f;
        screen.pMax.x = 1.f;
        screen.pMin.y = -1.f / frame;
        screen.pMax.y = 1.f / frame;
    }
    //默认视场角是90度。
    float fov = 90.0f;
    float halffov = 45.0f;
    return new PerspectiveCamera(cam2world, screen, lensradius,
        focaldistance, fov);
}
```

3.2 相机测试

等相机移植好以后，我们就可以开始测试了。

在场景文件加载时（api.cpp），pbrtLookAt 函数会调用参数执行：

```
    Transform lookAt =
        LookAt(Point3f(ex, ey, ez), Point3f(lx, ly, lz), Vector3f(ux, uy, uz
            ));
    FOR_ACTIVE_TRANSFORMS(curTransform[i] = curTransform[i] * lookAt;);
```

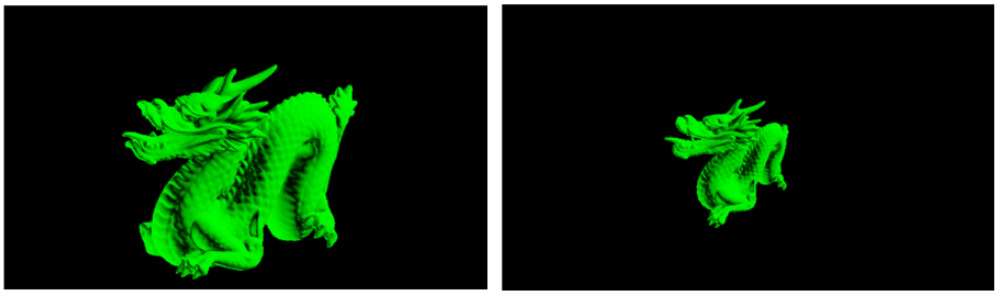
curTransform[i] 是全局变换。如果场景文件中没有定义全局的变换时，里面就是单位矩阵，则 curTransform[i] 就是 lookAt 返回的矩阵了。lookAt 返回的矩阵是将世界坐标系变换到相机坐标系。pbrtCamera() 函数（api.cpp）会调用 renderOptions->CameraToWorld = Inverse(curTransform); 之后 RenderOptions::MakeCamera() 函数（api.cpp）会使用 CameraToWorld 来调用 pbrt::MakeCamera() 函数（api.cpp）来生成相机。

假如我们要生成的图是 500*300 分辨率的的图：

```
    Camera* cam;
    //初始化程序
        Point3f eye(-3.0f, 1.5f, -3.0f), look(0.0, 0.0, 0.0f);
        Vector3f up(0.0f, 1.0f, 0.0f);
        Transform lookat = LookAt(eye, look, up);
    //取逆是因为LookAt返回的是世界坐标到相机坐标系的变换
    //而我们需要相机坐标系到世界坐标系的变换
        Transform Camera2World = Inverse(lookat);
        cam = CreatePerspectiveCamera(Camera2World);
    //渲染程序
        for (int j = 0; j < 300; j++) {
                for (int i = 0; i < 500; i++) {
                        CameraSample cs; cs.pFilm = Point2f(i + random(), j
                            + random());
                        Ray r;
                        cam->GenerateRay(cs,&r);
                        float tHit;
```

```
17                          SurfaceInteraction isect;
18                          Spectrum colObj(0.0f);
19                          if (agg->Intersect(r, &isect)){
20                                  float Li = Dot(Light, isect.n);
21                                  //if (Li < 0)Li = 0;
22                                  colObj[1] = std::abs(Li);
23                          }
24              //更新像素值
25              //......
26                  }
27          }
```

我们分别调用生成正交投影相机和透视投影相机的两个程序，渲染出的结果如下，左图是正交投影的结果，右图是透视投影的结果。



可能您会在移植的过程中遇到各种 Bug，最后显示的结果有问题，这个时候您需要多检查几次。值得庆幸的是，我们的程序现在还不是很复杂，所以调试起来还算容易。相机和各种变换之间的关系大家一定要认真对照着源码看几遍，一定要把各个变换关系牢记于心，这样以后再遇到与变换有关的内容，就能学起来得心应手了。

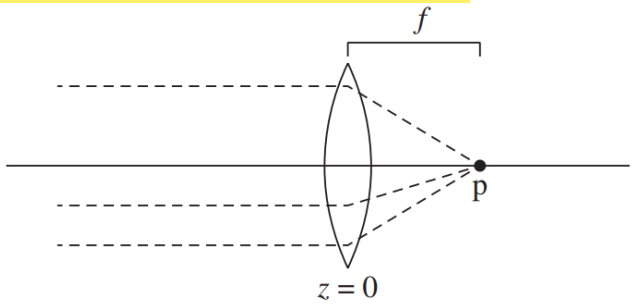
四　薄透镜模型和景深

本章是本书最难的一章（相对前面的内容，本节算比较难的了），本章的意义在于实现离焦模糊效果。

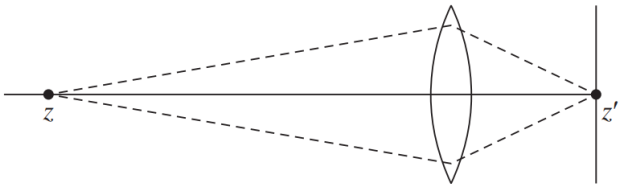
4 1　PBRT 薄镜头模拟

对于针孔相机，小光圈在实际世界中，需要更长的曝光时间来得到足够的光子，因此容易发生动态模糊现象。而大光圈时，镜头仅能聚焦于某个平面（我们称为焦平面）。

对于目前介绍的简单相机模型，我们可以应用光学中的经典近似，即薄透镜近似来模拟光圈的影响。在薄透镜近似下，通过透镜的平行入射光线聚焦在透镜后面的一个点上，称为焦点，如下图。焦点在透镜后面的距离 f 是透镜的焦距。也就是说如果胶卷平面的距离等于镜头后面的焦距，那么无限远处的物体就会聚焦，因为它们会成像到胶卷上的一个点。



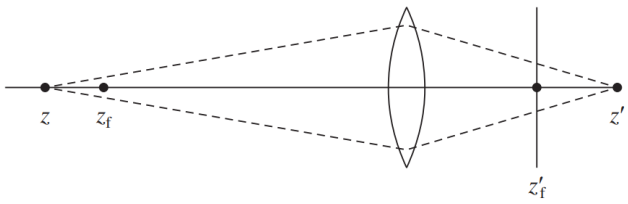
要聚焦某个平面上的点，聚焦面离透镜的距离就有不同：



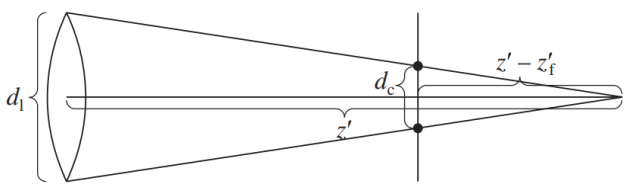
它们满足一个关系：

$$\frac{1}{z'} - \frac{1}{z} = \frac{1}{f} \tag{四.1}$$

也就是说当 $z = -\infty$ 时，我们得到 $z' = f$，符合上面的结论。不在焦点平面上的点被成像到胶片平面上的圆盘上，而不是成像到一个点上。这个圆盘的边界称为混淆圆。混淆圆的大小受光线穿过的光圈直径、焦距以及物体与透镜之间的距离的影响。距离镜头的物体出现在焦点上的范围，称为镜头的景深。我们可以用上面的公式计算，假如一个焦距为 f 的镜头，聚焦距离镜头为 z_f 处的点，聚焦胶片平面在 z'_f；当给另一个点在位置 z 时，它会被聚焦到 z' 的位置上，要么在当前焦平面的后面，要么在其前面。

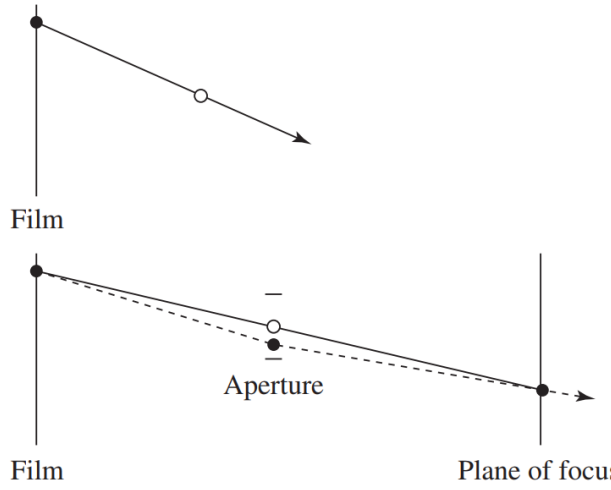


对于混淆圈，我们可以根据三角形法则求出其半径：



大家可以看 PBRT 书 [1] 第六章找到相应的公式。

在光线跟踪器中建立一个薄透镜的模型非常简单：所需要做的就是在透镜上选择一个点，然后找到合适的光线，在这个点上从透镜开始，这样焦点平面上的物体就可以聚焦在胶片上：



上图的上面表示的是我们以前的针孔相机，都是从胶片上发出光线。而上图的下面模拟的薄镜头相机，首先选择一个光圈，然后再光圈上选择中心点和另外一个点，我们假设 Film 上通过这两个点的 Ray 正好聚焦在 Film 上，因此计算这两个 Ray 在远平面的交点即可。之后再根据公式就能计算出我们要聚焦的平面的距离了。实际操作中，我们为每条光线在透镜上采样一个点，然后我们计算穿过透镜中心的光线（对应于针孔模型）和与焦点平面相交的点（实线）。我们知道，无论镜头样本的位置如何，焦点平面上的所有物体都必须对焦。因此，对应于镜头位置样本（虚线）的光线由从镜头样本点开始并通过聚焦平面上的计算交点的光线给出。

焦点平面（Plane of focus）是 PBRT 上的一种写法，区别于 film，表示能够与镜头一定距离，能够聚焦在 film 上的平面。

4 2 移植与测试

移植和测试就简单多了。ProjectiveCamera 不用管，我们之前保留了参数 lensr 和 focald，现在移植就很方便了。

对于 OrthographicCamera 和 PerspectiveCamera 两种相机，产生景深的程序都是一样的，我们需要修改 GenerateRay 函数：

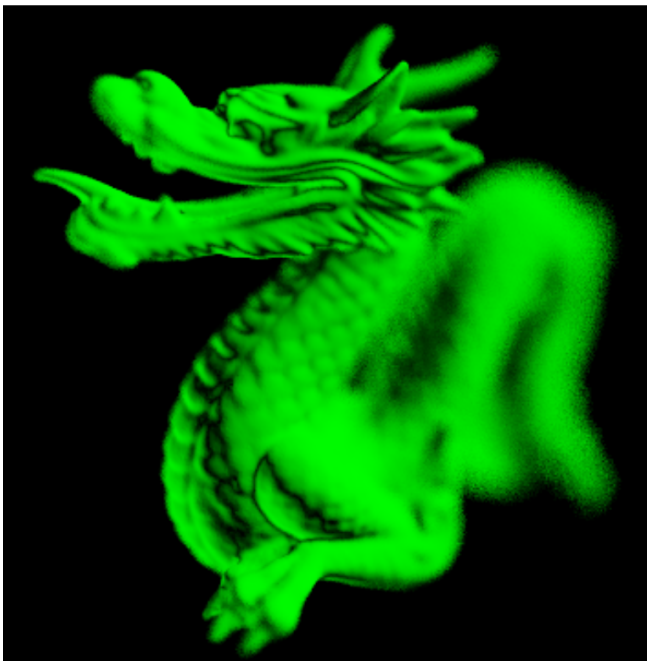
```
1    if(lensRadius > 0){
2        //在lens上采样一些点
3        ......
4        //计算聚焦平面（Plane of focus）
5        ......
6        //更新Ray
7        ......
8    }
```

我们对照源码，将该函数复制过去。注意 ConcentricSampleDisk 函数，该函数从两个 [0,1) 随机数生成一个圆心在 (0,0) 位置的单位圆盘上的随机数。

之后我们在 CreatePerspectiveCamera 里修改景深和光圈大小。我设置的景深为 3.0，光圈为 0.3（根据场景比例，这个光圈已经设置的很大了），注意不要忘记修改 CameraSample 对象：

```
1    CameraSample cs;
2    cs.pFilm = Point2f(i + random(), j + random());
3    cs.pLens = Point2f(random(), random());
```

显示效果如下：



至此，PBRT 的相机系统我们就学完了。

五　本书结语

这本小书只用了一天就完成了写作，毕竟内容确实不多。

之前说过，全景相机太简单暂时用不到，我们就不学了。真实感相机需要模拟一堆镜头的效果，太麻烦而且暂时也用不到，我们也不学了。建议如果各位想学的话，可以先研究研究实际的相机系统，找个相机入门手册，或者摄影入门课，然后再看 PBRT 书 [1] 就会容易很多。千万别一上来就看 [1]，不然很可能会一头雾水。

关于下一本书我要写什么内容也暂时没有拿定注意。除了采样重建相对容易一些（但是需要读者有一定基础，比如傅里叶变换，Nyquist 频率），其他几个可以开始讲述的章节，例如光线微分、纹理与反混淆、反射与材质等，每个都比较复杂和困难。大家阅读完前几本书就会发现，我们在学习过程中剥离了很多内容，但是要知道，欠的债总会还的——剥离的内容我们会在以后的学习中不断补充上去。

那么，下本书再见啦！

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] Marschner S, Shirley P. Fundamentals of computer graphics[M]. CRC Press, 2018.