Physically Based Rendering读书笔记 part 5

计算机图形学 pbrt

by silver_gp
原文地址：http://www.pbr-book.org/3ed-2018/contents.html

6 Camera Models

本章依然主要是讲代码。
之前介绍了针孔摄像机，但是它是一个简化模型，忽略了光线进入和离开镜头时的效果。摄像机镜头系统引入了一系列假设，能影响到最终图片的效果：vignetting：边缘发黑中间更亮；pincushion或者barrel distortion：能让直线在镜头当中显得扭曲。
本章介绍Camera类，它有两个重要的方法：Camera::GenerateRay()和Camera::GenerateRayDifferential()。前者生成世界空间内的射线，用于从投影面中采样。基于图片的不同格式来生成不同的采样射线，pbrt可以创造出不同类型的图片用来渲染3d场景。后者不但生成射线，还会包含了图片采样区域的额外信息在，这些信息用于抗锯齿。还有一些额外方法用于支持双向光线传输算法(bidirectional light transport algrithms)。

6.1 Camera Model

首先定义camera接口类

```
<<Camera Declarations>>=
class Camera {
public:
    <<Camera Interface>>
    <<Camera Public Data>>
};
```

```
<<Camera Interface>>=
Camera(const AnimatedTransform &CameraToWorld, Float shutt
erOpen,
       Float shutterClose, Film *film, const Medium *mediu
m);
<<Camera Public Data>>=
///camera在场景中的姿态
AnimatedTransform CameraToWorld;
///开关镜头的时间，用于计算motion blur
const Float shutterOpen, shutterClose;
///输出图片film
Film *film;
///散射介质，以后会讲
const Medium *medium;
```

Camera的派生类首先要实现的方法是

```
//函数的返回值代表击中film的光线的radiance的量，简单的camera都返
回1，比较复杂的摄像机会返回别的值
<<Camera Interface>>+=
virtual Float GenerateRay(const CameraSample &sample,
                          Ray *ray) const = 0;
```

需要注意的是，返回的射线的Direction，必须是**归一化**的。

```
<<Camera Declarations>>+=
struct CameraSample {
    ///射线击中film上的目标点
    Point2f pFilm;
    ///射线穿过镜头的点
    Point2f pLens;
    ///射线采样场景的时间，这涉及到camera的shutterOpen-shutterC
lose时间范围，通过这个时间进行线性插值。
    Float time;
};
```

```
<<Camera Method Definitions>>=
Float Camera::GenerateRayDifferential(const CameraSample &
sample,
        RayDifferential *rd) const {
    //首先要完成GenerateRay的功能
    Float wt = GenerateRay(sample, rd);
    //然后要在目标投影面上，x和y方向各偏移一个像素，计算射线
    //用于取得一个信息：一条射线采样的结果，到底代表了film上的多大
面积
    //通常用于抗锯齿
    <<Find camera ray after shifting one pixel in the x di
rection>>
    <<Find camera ray after shifting one pixel in the y di
rection>>
    rd->hasDifferentials = true;
    return wt;
}
```

```
<<Find camera ray after shifting one pixel in the x direct
ion>>=
CameraSample sshift = sample;
//将sample在film上的x像素+1
sshift.pFilm.x++;
//然后生成射线
Ray rx;
Float wtx = GenerateRay(sshift, &rx);
if (wtx == 0) return 0;
rd->rxOrigin = rx.o;
rd->rxDirection = rx.d;
```

6.1.1 Camera Coordinate Spaces

之前介绍过物体空间(object space)和世界空间(world space)，现在介绍摄像机空间(camera space)。

- 物体空间(object space)：定义几何图元的空间
- 世界空间(world space)：每个图元有自己的物体空间，所有物体在场景中，都放置在世界空间中。每个图元都有物体到世界空间的转换。世界空间是所有用于定义其他空间的标准框架。
- 摄像机空间(camera space)：摄像机放置于世界空间中，它有特定的方向和朝向。camera定义了一个新的坐标系，其原点就是摄像机的位置，这个坐标系的z轴对应的摄像机的朝向，y轴对应摄像机的上方向。

6.2 Projective Camera Models

场景渲染最基础的问题，是如何把3D场景投影到2D图像上，毫无疑问我们用的是4x4的投影矩阵，因此就有了这个ProjectiveCamera类，这下面又分为两种，一种是正交投影(orthographic projection)，另一种是透视投影(perspective projection)。

```
<Camera Declarations>>+=
class ProjectiveCamera : public Camera {
public:
    <<ProjectiveCamera Public Methods>>
protected:
    <<ProjectiveCamera Protected Data>>
};
```

在讨论投影摄像机，有几个坐标系是非常有用的：

- 屏幕空间(Screen Space)：在这里，屏幕空间定义在投影面(film plane)上的。摄像机将物体从摄像机空间投影到投影面上，其中在屏幕窗口(screen window)范围内的内容会被生成为图像。深度z的范围在[0,1]，相关的点在近裁减平面和远裁减平面之间。
- 归一化设备坐标空间(Normalized device coordinate space, NDC space)：这个坐标系是真正用来渲染的坐标系， x和y的范围都是从$(0,0)$到$(1,1)$，其中$(0,0)$是图像左上角。深度值与屏幕空间相同。
- 光栅化空间(raster space)：与NDC空间几乎相同，不过x和y坐标范围是$(0,0)$到$(resolution.x, resolution.y)$。

投影摄像机用4×4矩阵来进行坐标转换，但是有些奇怪的图片特性就无法仅仅用矩阵变换来描述了。

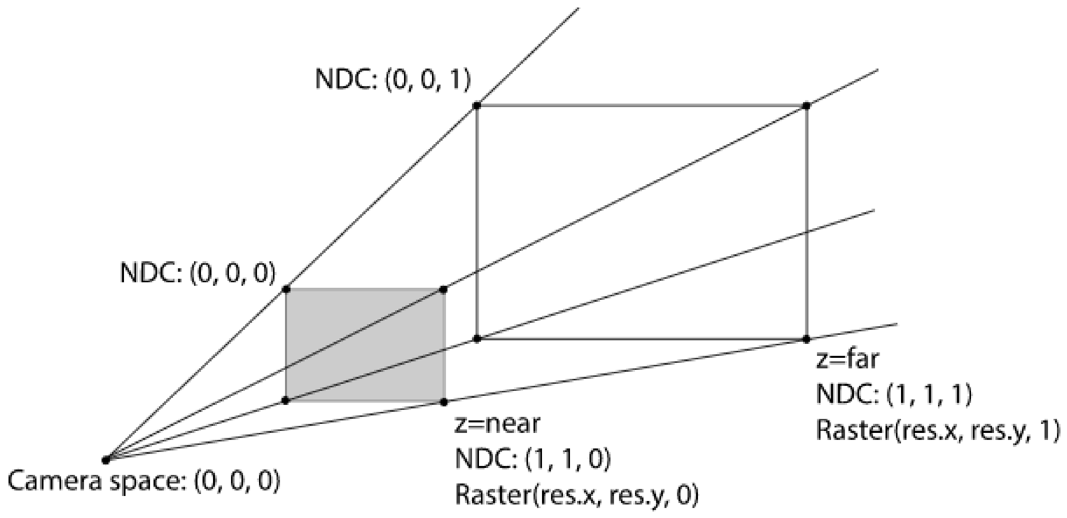

Figure 6.1: Several camera-related coordinate spaces are commonly used to simplify the implementation of `Camera`s. The camera class holds transformations between them. Scene objects ir world space are viewed by the camera, which sits at the origin of camera space and points along the $+z$ axis. Objects between the near and far planes are projected onto the film plane at $z = \mathrm{near}$ in camera space. The film plane is at $z = 0$ in raster space, where x and y range from $(0, 0)$ to $(\mathrm{resolution.\,x}, \mathrm{resolution.\,y})$. Normalized device coordinate (NDC) space normalizes raster space so that x and y range from $(0, 0)$ to $(1, 1)$.

```
//构造函数还需要一个投影矩阵、屏幕尺寸，以及dof用的其他参数
<<ProjectiveCamera Public Methods>>=
ProjectiveCamera(const AnimatedTransform &CameraToWorld,
        const Transform &CameraToScreen, const Bounds2f &s
creenWindow,
        Float shutterOpen, Float shutterClose, Float lens
r, Float focald,
        Film *film, const Medium *medium)
    : Camera(CameraToWorld, shutterOpen, shutterClose, fil
m, medium),
      CameraToScreen(CameraToScreen) {
    <<Initialize depth of field parameters>>
    <<Compute projective camera transformations>>
}
```

```
<<Compute projective camera transformations>>=
<<Compute projective camera screen transformations>>
//构造了矩阵raster-->screen-->camera
RasterToCamera = Inverse(CameraToScreen) * RasterToScreen;
<<ProjectiveCamera Protected Data>>=
Transform CameraToScreen, RasterToCamera;
```

```
<<Compute projective camera screen transformations>>=
ScreenToRaster = Scale(film->fullResolution.x,
                       film->fullResolution.y, 1) *   //3.最后再次缩放坐标到光栅化分辨率
    Scale(1 / (screenWindow.pMax.x - screenWindow.pMin.x),
          1 / (screenWindow.pMin.y - screenWindow.pMax.y),
 1) * //2.将screen window尺寸缩放到[0,1]之间，注意这里y反向了
    Translate(Vector3f(-screenWindow.pMin.x, -screenWindow.pMax.y, 0));//1.首先将screen window的左上角坐标置与原点
//逆矩阵
RasterToScreen = Inverse(ScreenToRaster);
<<ProjectiveCamera Protected Data>>+=
Transform ScreenToRaster, RasterToScreen;
```

6.2.1 Orthographic Camera

```
<<OrthographicCamera Declarations>>=
class OrthographicCamera : public ProjectiveCamera {
public:
    <<OrthographicCamera Public Methods>>
private:
    <<OrthographicCamera Private Data>>
};
```

正交摄像机(orthographics camera)是基于正交投影变换(orthographics projection transformation)。正交变换将场景中的一个立方体区域投影到这个盒子的前面。它不会有近大远小的投影效果，平行线依然是平行的，并且保持物体间的相对距离保持不变。

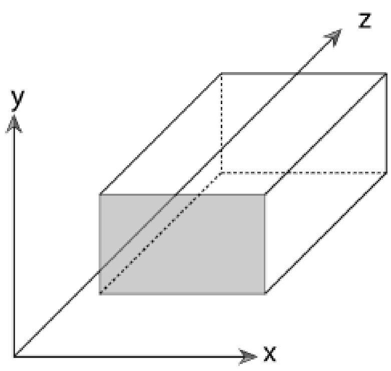

Figure 6.2: The orthographic view volume is an axis-aligned box in camera space, defined such that objects inside the region are projected onto the $z =$ near face of the box.

```
<<OrthographicCamera Public Methods>>=
OrthographicCamera(const AnimatedTransform &CameraToWorld,
        const Bounds2f &screenWindow, Float shutterOpen,
        Float shutterClose, Float lensRadius, Float focalD
istance,
        Film *film, const Medium *medium)
        //调用基类ProjectiveCamera的构造函数，但Camera->Scree
n传的是Orthographics矩阵
    : ProjectiveCamera(CameraToWorld, Orthographic(0, 1),
                       screenWindow, shutterOpen, shutterC
lose,
                       lensRadius, focalDistance, film, me
dium) {
    <<Compute differential changes in origin for orthograp
hic camera rays>>
}
```

```
<<Transform Method Definitions>>+=
Transform Orthographic(Float zNear, Float zFar) {
    //正交投影是保留x,y坐标而缩放z坐标到[0,1]范围内
    return Scale(1, 1, 1 / (zFar - zNear)) *
           Translate(Vector3f(0, 0, -zNear));
}
```

```
//正交投影下，微分射线的方向与主射线方向一致，不同的是起点是pilm pl
ane的xy偏移一个像素
//这里计算了raster space偏移一个像素后的起点偏移，转换到camera sp
ace下
//正交投影这么做就够了，因为不存在透视的问题
<<Compute differential changes in origin for orthographic
 camera rays>>=
dxCamera = RasterToCamera(Vector3f(1, 0, 0));
dyCamera = RasterToCamera(Vector3f(0, 1, 0));
<<OrthographicCamera Private Data>>=
Vector3f dxCamera, dyCamera;
```

一个raster space下的点要转换为摄像机射线(camera ray)。如下图。首先raster space下的采样点转换到camera space，这个点在近投影平面上，它就是射线的起点，由于是

在camera空间下，那么显然的方向就是z轴正方向$(0, 0, 1)$。

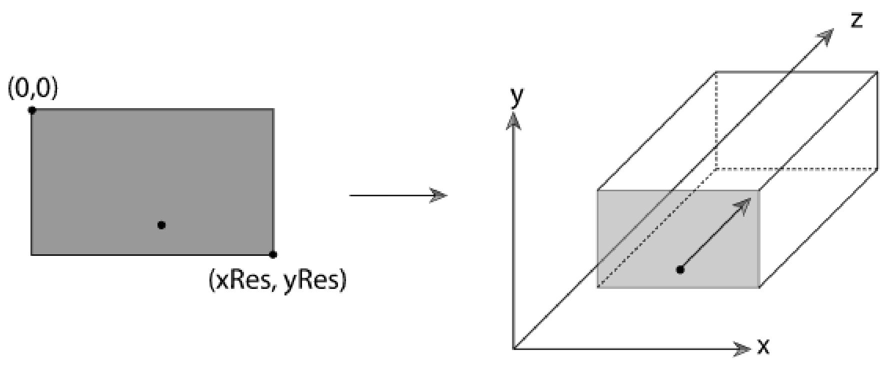

Figure 6.4: To create a ray with the orthographic camera, a raster space position on the film plane is transformed to camera space, giving the ray's origin on the near plane. The ray's direction in camera space is $(0, 0, 1)$, down the z axis.

```
<<OrthographicCamera Definitions>>=
Float OrthographicCamera::GenerateRay(const CameraSample &
sample,
        Ray *ray) const {
    <<Compute raster and camera sample positions>>
    *ray = Ray(pCamera, Vector3f(0, 0, 1));
    //如果开启了景深效果(Depth Of Field, DOF)，那么射线的起点和
方向都会有一定的修改。
    <<Modify ray for depth of field>>
    //传入的sample.time是[0,1)区间内的，因此ray->time是shutter
Open和shutterCLose的插值结果
    ray->time = Lerp(sample.time, shutterOpen, shutterClos
e);
    ray->medium = medium;
    *ray = CameraToWorld(*ray);
    return 1;
}
```

```
<<Compute raster and camera sample positions>>=
//矩阵准备好以后，就可以很简单的吧film的坐标转换为camera空间下的坐
标
Point3f pFilm = Point3f(sample.pFilm.x, sample.pFilm.y,
0);
Point3f pCamera = RasterToCamera(pFilm);
```

```
///生成微分射线
<<OrthographicCamera Definitions>>+=
Float OrthographicCamera::GenerateRayDifferential(
        const CameraSample &sample, RayDifferential *ray)
const {
    <<Compute main orthographic viewing ray>>
    //根据正交摄像机参数生成初始的起始点和方向
    <<Compute ray differentials for OrthographicCamera>>
    //这部分跟生成普通射线一样
    ray->time = Lerp(sample.time, shutterOpen, shutterClos
e);
    ray->hasDifferentials = true;
    ray->medium = medium;
    *ray = CameraToWorld(*ray);
    return 1;
}
<<Compute ray differentials for OrthographicCamera>>=
if (lensRadius > 0) {
    //这部分代码跟透镜参数有关，后面会讲
    <<Compute OrthographicCamera ray differentials account
ing for lens>>
} else {
    //在原始射线的基础上，起始点加1个像素的偏移
    ray->rxOrigin = ray->o + dxCamera;
    ray->ryOrigin = ray->o + dyCamera;
    //当然方向还是原来的方向，正交摄像机射线方向不用改
    ray->rxDirection = ray->ryDirection = ray->d;
}
```

6.2.2 Perspective Camera

透视投影与正交投影类似，将一个空间映射到一个2D 投影面上。然而它有一个特性：近大远小。

```
<<PerspectiveCamera Declarations>>=
class PerspectiveCamera : public ProjectiveCamera {
public:
    <<PerspectiveCamera Public Methods>>
private:
    <<PerspectiveCamera Private Data>>
};
<<PerspectiveCamera Method Definitions>>=
PerspectiveCamera::PerspectiveCamera(
        const AnimatedTransform &CameraToWorld,
        const Bounds2f &screenWindow, Float shutterOpen,
        Float shutterClose, Float lensRadius, Float focalD
istance,
        Float fov, Film *film, const Medium *medium)
    : ProjectiveCamera(CameraToWorld, Perspective(fov, 1e-
2f, 1000.f),
                       screenWindow, shutterOpen, shutterC
lose,
                       lensRadius, focalDistance, film, me
dium) {
    <<Compute differential changes in origin for perspecti
ve camera rays>>
    <<Compute image plane bounds at  for PerspectiveCamera
>>
}
```

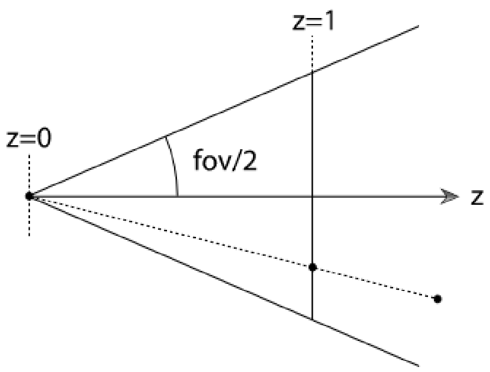

Figure 6.5: The perspective transformation matrix projects points in camera space onto the film plane. The x' and y' coordinates of the projected points are equal to the unprojected x and y coordinates divided by the z coordinate. The projected z' coordinate is computed so that points on the near plane map to $z' = 0$ and points on the far plane map to $z' = 1$.

如图所示，透视投影将场景投影到投影平面(viewing plane)，这个投影平面与z轴垂直。Perspective()函数接受张角fov，远裁减面和近裁减面这几个参数。在投影变换之后，在近裁减面的点的$z = 0$，在远裁减面的$z = 1$。渲染系统是基于光栅化的，因此设置这些裁减面要非常小心，它们定义了场景渲染的z的范围，同时它可能造成精度上的错误。(原文是 with too many orders of magnitude variation between their values can lead to numberical precision errors)。在pbrt中我们可以任意定义它。

```
<<Transform Method Definitions>>+=
Transform Perspective(Float fov, Float n, Float f) {
    <<Perform projective divide for perspective projection
>>
    <<Scale canonical perspective view to specified field
 of view>>
}
```

变换分为两步：

1. 将camera空间的点P，变换到显示平面(viewing plane)。

$$
\begin{aligned}
x' &= x/z \\
y' &= y/z \\
z' &= \frac{f(z-n)}{z(f-n)}
\end{aligned}
$$

其矩阵形式为：

$$
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & \frac{f}{f-n} & -\frac{fn}{f-n} \\
0 & 0 & 1 & 0
\end{bmatrix}
$$

```
<<Perform projective divide for perspective projection>>=
Matrix4x4 persp(1, 0,           0,                0,
                0, 1,           0,                0,
                0, 0, f / (f - n), -f*n / (f - n),
                0, 0,           1,                0);
```

2. fov(field of view)角度用于缩放(x, y)坐标到投影平面，这样一来在fov内的投影平面内点的坐标就被限制到了[-1,1]，注意它是同时作用于x和y坐标。对于方形图像来说，x和y坐标在screen space都是$[-1, 1]$范围内。但对于非正方形的图像而言，比较窄的那个方向被映射到$[-1, 1]$，而比较宽的那个方向必然就映射到了比较大的范围。回想正切值就是对边比临边，这里，临边长度就是1，因此对边的长度为$tan(fov/2)$，把这个范围要对应地映射到$[-1, 1]$内。**用简单的话来说，fov就是用来缩放xy坐标的，因为投影面是在z=1处，因此对于投影面上任一个坐标轴上坐标为tan(fov/2)的点，只有乘以1 / tan(fov / 2)才能缩放到1，因此任何坐标经过persp转换后都要进行一次这样的缩放，为了归一化。**

```
<<Scale canonical perspective view to specified field of v
iew>>=
Float invTanAng = 1 / std::tan(Radians(fov) / 2);
return Scale(invTanAng, invTanAng, 1) * Transform(persp);
```

```
//计算微分射线的起点
<<Compute differential changes in origin for perspective c
amera rays>>=
dxCamera = (RasterToCamera(Point3f(1, 0, 0)) -
            RasterToCamera(Point3f(0, 0, 0)));
dyCamera = (RasterToCamera(Point3f(0, 1, 0)) -
            RasterToCamera(Point3f(0, 0, 0)));
<<PerspectiveCamera Private Data>>=
Vector3f dxCamera, dyCamera;
```

```
<<PerspectiveCamera Method Definitions>>+=
Float PerspectiveCamera::GenerateRay(const CameraSample &s
ample,
        Ray *ray) const {
    //射线方向就是原点 指向 pCamera的位置，pCamera是film上一点转
换到camera space
    <<Compute raster and camera sample positions>>
    *ray = Ray(Point3f(0, 0, 0), Normalize(Vector3f(pCamer
a)));
    <<Modify ray for depth of field>>
    ray->time = Lerp(sample.time, shutterOpen, shutterClos
e);
    ray->medium = medium;
    *ray = CameraToWorld(*ray);
    return 1;
}
```

```
<<PerspectiveCamera Public Methods>>=
Float GenerateRayDifferential(const CameraSample &sample,
                              RayDifferential *ray) const;
<<Compute offset rays for PerspectiveCamera ray differenti
als>>=
if (lensRadius > 0) {
    <<Compute PerspectiveCamera ray differentials accounti
ng for lens>>
} else {
    ///微分射线就如定义
    ray->rxOrigin = ray->ryOrigin = ray->o;
    ray->rxDirection = Normalize(Vector3f(pCamera) + dxCam
era);
    ray->ryDirection = Normalize(Vector3f(pCamera) + dyCam
era);
}
```

6.2.3 The Thin Lens Model and Depth of Field

理想的针孔摄像机只允许光线通过单点抵达胶片，但这并不符合真实的物理现象。虽然可以通过制造出有极小小孔的摄像机来达到这个效果，但空洞太小使得光线进入得太少，这就不得不需要延长曝光时间来获得图像，但这又会反过来使得在照相机快门打开时，场景运动物体变得模糊。
真实的摄像机设计会遇到一个权衡：更大的光圈孔径(aperture)，会有更多的光线到达胶片，并且可以有更少的曝光时间。然而镜头只能聚焦于一个单独的平面（焦平面），场景远处的物体就会不可避免的模糊。更大的光圈孔径会使得与焦点不同深度的物体产生不同程度的模糊。
作为一个简单的摄像机模型，我们只要使用经典的假设就可以了：薄透镜近似（thin lens approximation）。薄透镜近似建模了一个光学系统，这个光学系统有一个独立的镜头，并且这个镜头是球形的，它的厚度相比于曲率半径非常小。
薄透镜近似下，入射光与光轴（optical axis）平行，并且入射光通过了透镜聚焦在一点上，那个点叫做焦点（focal point），焦点与镜头的距离f，成为镜头的焦距（focal length）。如果投影面放在焦点上，物体距离镜头无限远的话，就会汇聚到焦点上，也就是说这个物体最终在投影面上的像就是一个点。
下图中，我们把镜头垂直于z轴，并且位置是$z = 0$，并且场景沿着$-z$方向（与之前实现的camera space不同，那里场景是沿着z轴正方向的）。因此在场景一侧的，与镜头的距离记为非素数变量z（负数），在投影面一侧的距离（正数)记为素数变量z'。

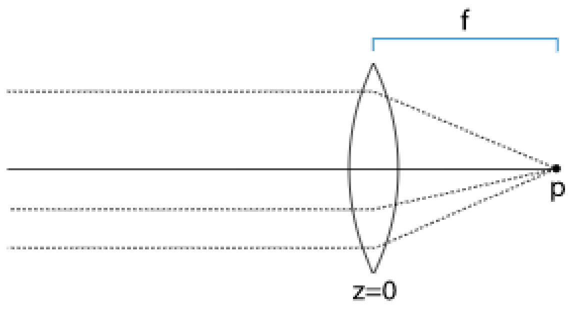

Figure 6.6: A thin lens, located along the z axis at $z = 0$. Incident rays that are parallel to the optical axis and pass through a thin lens (dashed lines) all pass through a point p, the focal point. The distance between the lens and the focal point, f, is the lens's focal length.

在场景一侧的点深度为z，焦距为f，那么高斯透镜公式(Gaussian lens equation)为：

$$\frac{1}{z'} - \frac{1}{z} = \frac{1}{f}$$

特别的，如果$z = -\infty$，那么显然$z' = f$。
根据高斯镜头方程，去求解**清晰成像下的镜头与投影面的距离**：

$$z' = \frac{fz}{f+z}$$

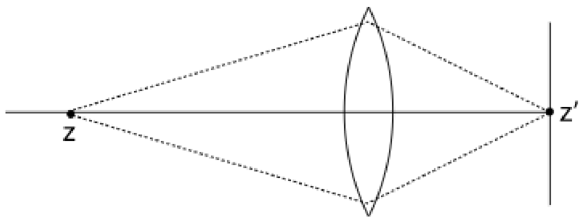

Figure 6.7: To focus a thin lens at a depth z in the scene, Equation (6.2) can be used to compute the distance z' on the film side of the lens that points at z focus to. Focusing is performed by adjusting the distance between the lens and the film plane.

通过这个公式，我们就能计算出场景侧任意一点可以成像到哪个位置上。记住，我们干的事，是指定一个焦平面z，就知道应该把film plane放到哪个位置上：z'。
注意，焦距f和焦平面(plane of focus)不是一个概念。焦平面指的是在场景侧，在这个平面上的所有点都能清晰成像。**焦距**指的是在无限远处发来的平行光可以汇聚到一个点上，这个点与镜头的距离。
因此上面那个公式，实际上给出的是**焦距f**，**焦平面z**，**投影面z'**三者的关系。
如果一个点没有落在焦平面上，那么它呈现出来的图像就是一个圆盘。这个圆盘的范围叫做模糊圈（circle of confusion）。这个尺寸与光线入射到镜头的直径、焦距以及物体到镜头的距离相关。下图展现了这个效果：

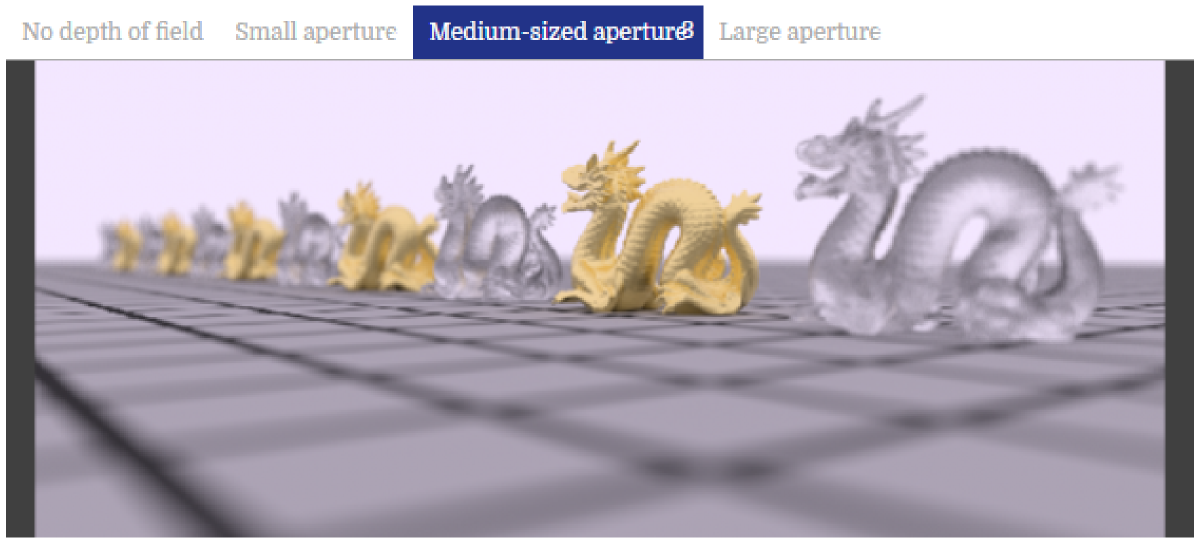

Figure 6.8: (1) Scene rendered with no depth of field, (2) depth of field due to a relatively small lens aperture, which gives only a small amount of blurriness in the out-of-focus regions, (3) and (4) As the size of the lens aperture increases, the size of the circle of confusion in the out-of-focus areas increases, giving a greater amount of blur on the film plane.

Figure 6.9: Depth of field gives a greater sense of depth and scale to this part of the landscape scene. (*Scene courtesy of Laubwerk.*)

实际上，物体不是精确地在焦平面上其实也能达到精准对焦的效果，因为模糊全太小了，比film的一个像素还低，这样的话，实际上物体也算是精准对焦的。因此这就有了这么个区间，在这个区间内，物体都是精准对焦的，这个区间就叫做镜头的**景深(depth of field)**。

高斯透镜方程也能帮助我们计算模糊圈。已知一个透镜，其焦距为f，在场景侧的距离为z_f处聚焦，此时投影面的距离为z'_f。给定另外一点的距离为z，高斯透镜方程计算出的投影面侧的距离为z'，如下图，这个点既可以在投影面也可以在投影面后面。

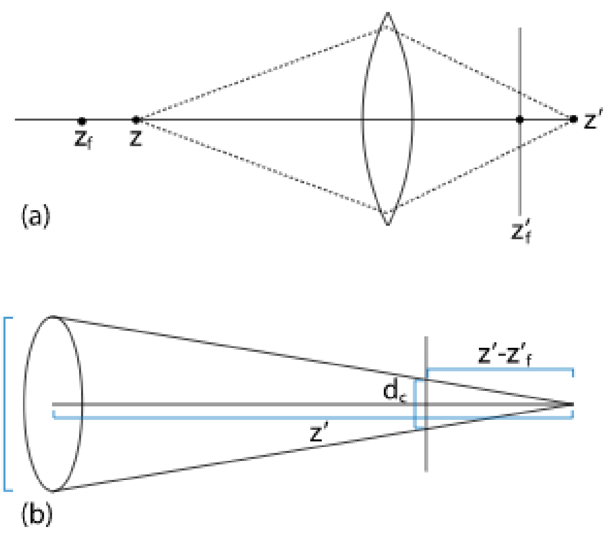

Figure 6.10: (a) If a thin lens with focal length f is focused at some depth z_f, then the distance from the lens to the film plane is z'_f, given by the Gaussian lens equation. A point in the scene at depth $z \neq z_f$ will be imaged as a circle on the film plane; here z focuses at z', which is behind the film plane. (b) To compute the diameter of the circle of confusion, we can apply similar triangles: the ratio of d_l, the diameter of the lens, to z' must be the same as the ratio of d_c, the diameter of the circle of confusion, to $z' - z'_f$.

如上图，我们设定的镜头焦平面在z_f处，因此投影面就在z'_f处，但是实际物体在z处，成的像本应在z'处，因此就在z_f的投影面上留下了个**模糊圈**。模糊圈的直径取决于z'点与投影面组成的锥形。如果已知镜头的直径为d_1，那么通过相似三角形能得出模糊圈直径d_c：

$$\frac{d_1}{z'} = \frac{d_c}{|z' - z'_f|}$$

因此，得出d_c：

$$d_c = \left| \frac{d_1(z' - z'_f)}{z'} \right|$$

通过高斯镜头方程，将投影面侧的距离换算为场景侧的距离，得到的结果为：

$$d_c = \left| \frac{d_1 f(z - z_f)}{z(f + z_f)} \right|$$

很显然的，我们能得到结论，模糊全的直径正比于镜头直径。这个参数经常被表述为f / n这样的分数形式，表示为焦距的几分之一，$d_1 = f/n$。
下图表示了50毫米的焦距，25毫米的镜头大小，聚焦距离为1米。可以看到，模糊效果在焦平面附近并不是均匀的，而是在焦平面前面的时候，增长很快，在后面的时候增长很慢。

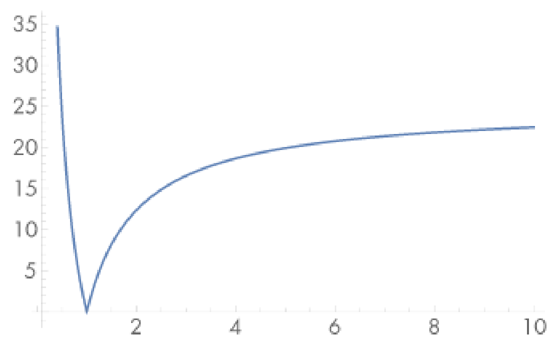

Figure 6.11: The diameter of the circle of confusion as a function of depth for a 50-mm focal length lens with 25-mm aperture, focused at 1 meter.

对薄透镜进行建模：选择在透镜上的一点，寻找合适的射线，从那一点开始出发指向焦平面上的点，如下图

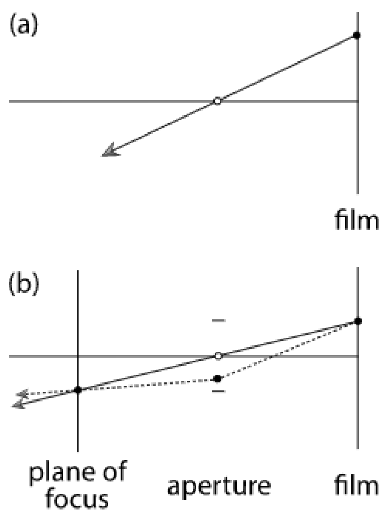

Figure 6.12: (a) For a pinhole camera model, a single camera ray is associated with each point on the film plane (filled circle), given by the ray that passes through the single point of the pinhole lens (empty circle). (b) For a camera model with a finite aperture, we sample a point (filled circle) on the disk-shaped lens for each ray. We then compute the ray that passes through the center of the lens (corresponding to the pinhole model) and the point where it intersects the plane of focus (solid line). We know that all objects in the plane of focus must be in focus, regardless of the lens sample position. Therefore, the ray corresponding to the lens position sample (dashed line) is given by the ray starting on the lens sample point and passing through the computed intersection point on the plane of focus.

从film上一点出发，经过针孔摄像机到达焦平面上的一点，从film上同样一点出发，经过有半径的镜头之后，到达焦平面上的一点，这两点实际上是同一点。因为焦平面上的苏搜友物体都可以清晰成像，它们都将汇聚到film上。因此，摄像机需要两个额外参数来实现景深效果：镜头的尺寸，以及焦平面的距离。

```
<<ProjectiveCamera Protected Data>>+=
Float lensRadius, focalDistance;
<<Initialize depth of field parameters>>=
lensRadius = lensr;
focalDistance = focald;
```

因此通常情况下，对于一个像素点需要进行多次采样才能实现景深效果，如下图，每个像素点会有4次采样：

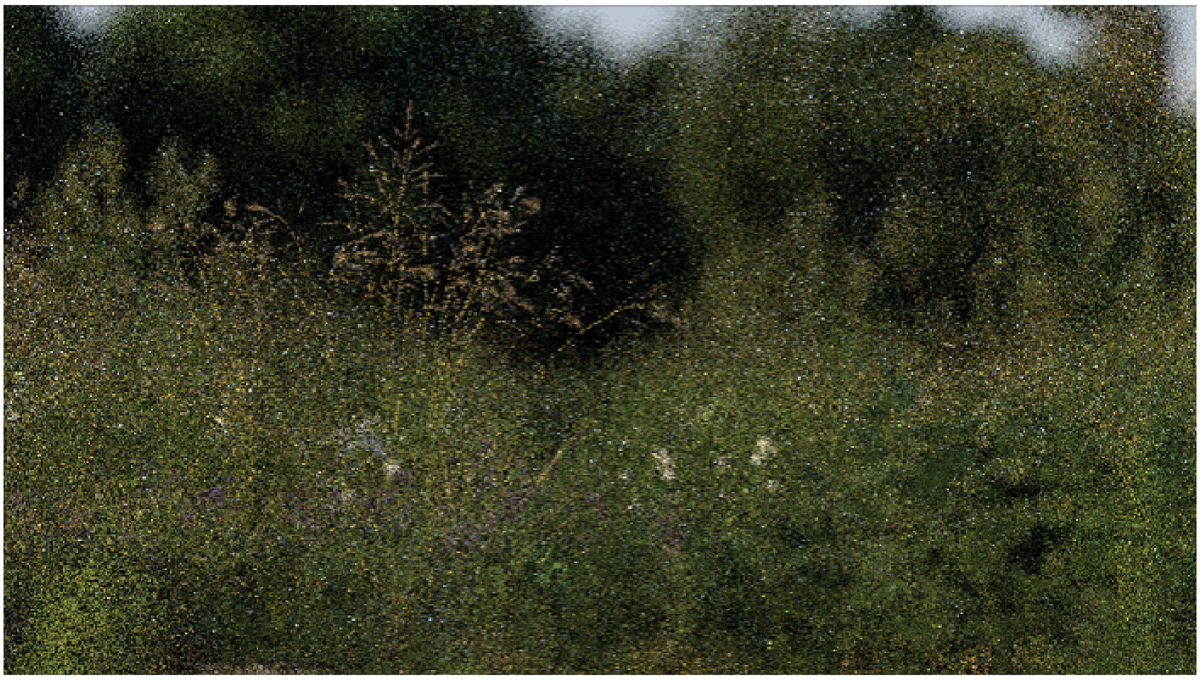

Figure 6.13: Landscape scene with depth of field and only four samples per pixel: the depth of field is undersampled and the image is grainy. (*Scene courtesy of Laubwerk.*)

```
<<Modify ray for depth of field>>=
if (lensRadius > 0) {
    <<Sample point on lens>>
    <<Compute point on plane of focus>>
    <<Update ray for effect of lens>>
}
```

ConcentrateSampleDisk()函数接受(u, v)坐标，将其映射到2D单位圆上，圆心在$(0, 0)$。

```
//得到镜头上的采样点，镜头半径 * 单位圆坐标
<<Sample point on lens>>=
Point2f pLens = lensRadius * ConcentricSampleDisk(sample.p
Lens);
```

有了这个镜头上的点了以后，就可以创造出一条射线。我们知道从焦平面上的点到镜头的光线，无论是镜头上的哪一点，都会汇聚到投影面的同一个点上，另外我们也知道通过镜头中心的射线是不会有方向的改变的。所以首先让投影面上某一点通过镜头中心，

与焦平面相交得到交点，然后在镜头上找一个采样点发射射线指向焦平面上的那个交点。
在这个模型中，焦平面与z轴垂直，射线的起点是原点，t就是交点位置：

$$t = \frac{focalDistance}{d_z}$$

```
<<Compute point on plane of focus>>=
Float ft = focalDistance / ray->d.z;
Point3f pFocus = (*ray)(ft);
```

接下来可以创建射线。起点是镜头上的一个采样点，重点就是这个交点pFocus。

```
<<Update ray for effect of lens>>=
ray->o = Point3f(pLens.x, pLens.y, 0);
ray->d = Normalize(pFocus - ray->o);
```

计算微分射线也是类似的办法， 不同之处在于会将投影平面两个方向各移动一个像素。

6.3 Environment Camera

光线跟踪相比于光栅化，有一个优点就是可以我们可以任意指定图像采样是如何映射到光线方向的，尽管大多数情况下，只是简单的直线。
接下来会引入一个新的摄像机模型，它追踪围绕场景中一个点的所有方向的光线，并且给出一个2D的显示。假设摄像机外有个球体，根据球体上任意一点都能生成一条射线。参数化表示，就是这个点是一个(θ, ϕ)对，其中$\theta \in [0, \pi]$，$\phi \in [0, 2\pi]$。这种图像的优点就是它表达了场景中某点的所有光线，可用作环境光。下图就是一个例子，θ从0（图像顶端）到π（图像底端），ϕ从0到2π：从图片的左边到右边。

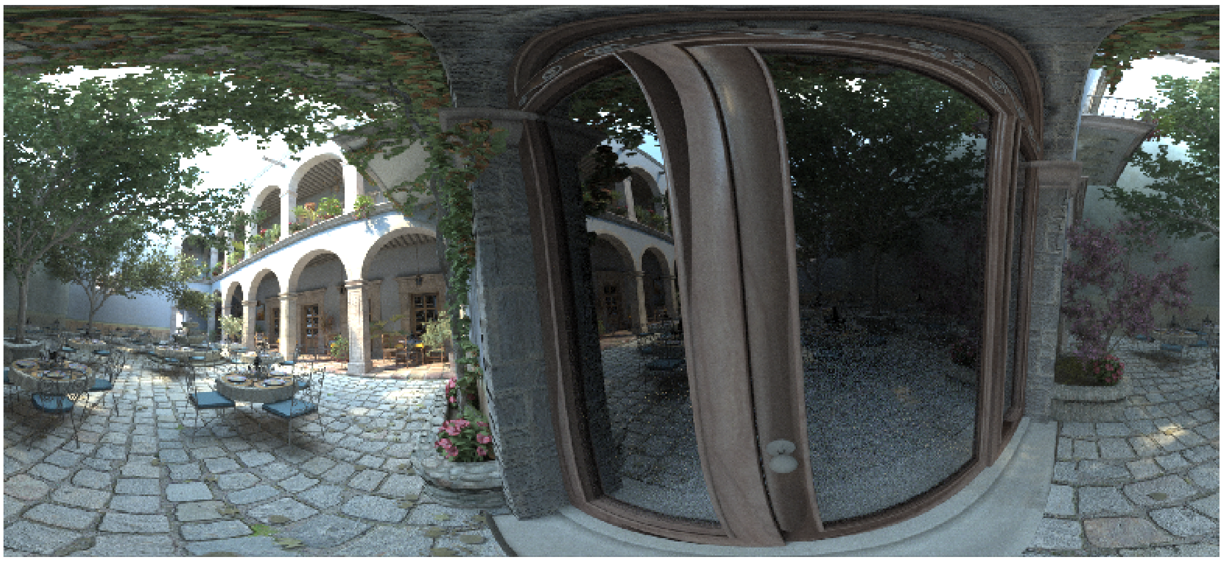

Figure 6.14: The San Miguel model rendered with the `EnvironmentCamera`, which traces rays in all directions from the camera position. The resulting image gives a representation of all light arriving at that point in the scene and can be used for the image-based lighting techniques described in Chapters 12 and 14.

EnvironmentCamera从Camera派生

```
<<EnvironmentCamera Public Methods>>=
EnvironmentCamera(const AnimatedTransform &CameraToWorld,
        Float shutterOpen, Float shutterClose, Film *film,
        const Medium *medium)
    : Camera(CameraToWorld, shutterOpen, shutterClose, fil
m, medium) {
}
<<EnvironmentCamera Method Definitions>>=
Float EnvironmentCamera::GenerateRay(const CameraSample &s
ample,
        Ray *ray) const {
    //根据theta,phi，计算出射线
    <<Compute environment camera ray direction>>
    *ray = Ray(Point3f(0, 0, 0), dir, Infinity,
               Lerp(sample.time, shutterOpen, shutterClos
e));
    ray->medium = medium;
    *ray = CameraToWorld(*ray);
    return 1;
}
```

```
<<Compute environment camera ray direction>>=
//将图片y坐标系映射到theta
Float theta = Pi * sample.pFilm.y / film->fullResolution.y;
//将图片x坐标映射到phi
Float phi = 2 * Pi * sample.pFilm.x / film->fullResolution.x;
//需要注意的是，camera空间下，y坐标向上为正，因此下面这个公式要符合这个方向
Vector3f dir(std::sin(theta) * std::cos(phi), std::cos(theta),
             std::sin(theta) * std::sin(phi));
```

6.4 Realistic Cameras

略