CHAPTER SIX

06 CAMERA MODELS

In Chapter 1, we described the pinhole camera model that is commonly used in computer graphics. This model is easy to describe and simulate, but it has a number of drawbacks. For example, everything rendered with a pinhole camera is in sharp focus, which can make images look computer generated. In order to make images that appear realistic, it is important to better simulate properties of real-world imaging systems.

Like the `Shapes` from Chapter 3, cameras in `pbrt` are represented by an abstract base class. This chapter describes the `Camera` class and its two closely related methods: `Camera::GenerateRay()` and `Camera::GenerateRayDifferential()`. The first method computes the world space ray corresponding to a sample position on the image plane. By generating these rays in different ways based on different models of image formation, the cameras in `pbrt` can create many types of images of the same 3D scene. The second method not only generates this ray but also computes information about the image area that the ray is sampling; this information is used for anti-aliasing computations in Chapter 10, for example. Here, we will show a few implementations of the `Camera` interface, each of which generates rays in a different way to simulate different imaging processes.

6.1 CAMERA MODEL

The abstract `Camera` base class holds generic camera options and defines the interface that all camera implementations must provide. It is defined in the files `core/camera.h` and `core/camera.cpp`.

⟨*Camera Declarations*⟩ ≡
```
class Camera {
public:
    ⟨Camera Interface 302⟩
    ⟨Camera Public Data 302⟩
};
```

The base `Camera` constructor takes several parameters that are appropriate for all camera types. One of the most important is the transformation that places the camera in the scene, which is stored in the `CameraToWorld` member variable. The `Camera` stores an `AnimatedTransform` to place itself in the scene (rather than just a regular `Transform`) so that the camera itself can be moving over time.

Real-world cameras have a shutter that opens for a short period of time to expose the film to light. One result of this nonzero exposure time is *motion blur*: objects that move during the film exposure time are blurred. If time values between the shutter open time and the shutter close time are associated with each ray, it is possible to compute images that exhibit motion blur. All `Camera`s therefore store a shutter open and shutter close time and are responsible for generating rays with appropriately set times. `Camera`s also contain an instance of the `Film` class to represent the final image to be computed. `Film` will be described in Section 7.8.

Camera implementations must pass along parameters that set these values to the `Camera` constructor. We will only show its prototype here because its implementation just copies the parameters to the corresponding member variables.

⟨*Camera Interface*⟩ ≡ 302
```
Camera(const AnimatedTransform &cam2world, float sopen, float sclose,
       Film *film);
```

⟨*Camera Public Data*⟩ ≡ 302
```
AnimatedTransform CameraToWorld;
const float shutterOpen, shutterClose;
Film *film;
```

The first method that camera subclasses need to implement is `Camera::GenerateRay()`, which generates a ray for a given image sample. The `CameraSample::imageX` and `CameraSample::imageY` components of the camera sample give the raster space x and y coordinates on the image plane for the ray (the complete contents of the `CameraSample` structure are described in detail in Chapter 7). It is important that the camera normalize the direction component of the returned ray—many other parts of the system will depend on this behavior.

This method also returns a floating-point value that gives a weight for how much light arriving at the film plane along the generated ray will contribute to the final image. Most cameras always return a value of one, but cameras that simulate real physical lens systems might need to set this value to indicate how much light the ray carries through the lenses based on their optical properties.

AnimatedTransform 96
Camera 302
Camera::GenerateRay() 303
CameraSample 342
CameraSample::imageX 342
CameraSample::imageY 342
Film 403
Transform 76

⟨*Camera Interface*⟩ +≡ 302
```
virtual float GenerateRay(const CameraSample &sample,
                          Ray *ray) const = 0;
```

The `GenerateRayDifferential()` method computes a main ray like `GenerateRay()` but also computes the corresponding rays for pixels shifted one pixel in the x and y directions on the image plane. This information helps give the rest of the system a notion of how much of the image area a particular camera ray's sample represents. Many `Camera` implementations will be able to compute these rays directly, but here we provide a default implementation that initializes shifted `CameraSample`s and calls the camera's `GenerateRay()` method repeatedly.

⟨*Camera Method Definitions*⟩ ≡
```
float Camera::GenerateRayDifferential(const CameraSample &sample,
                                      RayDifferential *rd) const {
    float wt = GenerateRay(sample, rd);
    ⟨Find ray after shifting one pixel in the x direction 303⟩
    ⟨Find ray after shifting one pixel in the y direction⟩
    if (wtx == 0.f || wty == 0.f) return 0.f;
    rd->hasDifferentials = true;
    return wt;
}
```

Finding the ray for one pixel over in x is just a matter of initializing a new `CameraSample` and copying the appropriate values into the `RayDifferential` structure. The implementation of the fragment ⟨*Find ray after shifting one pixel in the y direction*⟩ follows similarly and isn't included here.

⟨*Find ray after shifting one pixel in the x direction*⟩ ≡ 303
```
CameraSample sshift = sample;
++(sshift.imageX);
Ray rx;
float wtx = GenerateRay(sshift, &rx);
rd->rxOrigin = rx.o;
rd->rxDirection = rx.d;
```

Camera::GenerateRay() 303
CameraSample 342
CameraSample::imageX 342
Ray 66
Ray::d 67
Ray::o 67
RayDifferential 69
RayDifferential::
hasDifferentials 69
RayDifferential::
rxDirection 69
RayDifferential::rxOrigin 69

6.1.1 CAMERA COORDINATE SPACES

We have already made use of two important modeling coordinate spaces, object space and world space. We will now introduce four useful coordinate spaces related to the camera and imaging, summarized in Figure 6.1. All together, we now have the following:

- *Object space:* This is the coordinate system in which geometric primitives are defined. For example, spheres in `pbrt` are defined to be centered at the origin of object space.
- *World space:* While each primitive may have its own object space, all objects in the scene are placed in relation to a single world space. Each primitive has an object-to-world transformation that determines where it is located in world space. World space is the standard frame that all spaces are defined in terms of.

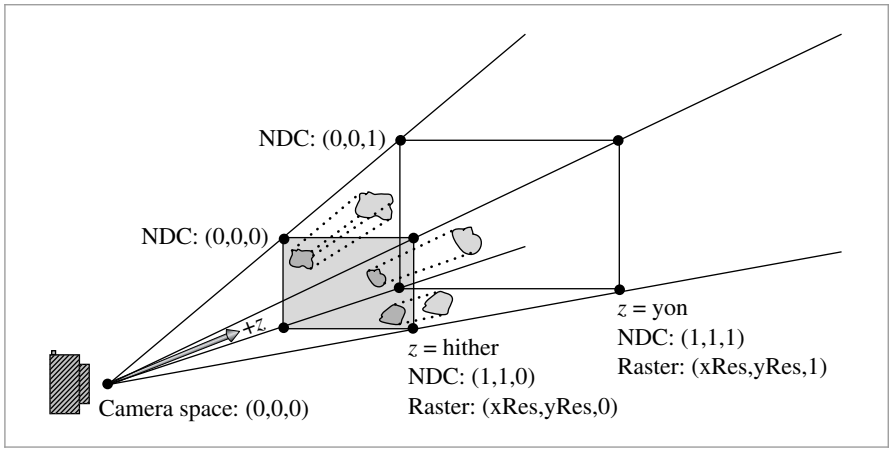


Figure 6.1: A handful of camera-related coordinate spaces help to simplify the implementation of `Camera`s. The camera class holds transformations between them. Scene objects in world space are viewed by the camera, which sits at the origin of camera space and points along the $+z$ axis. Objects between the hither and yon planes are projected onto the image plane at $z = \text{hither}$ in camera space. The image plane is at $z = 0$ in raster space, where x and y range from $(0, 0)$ to $(\text{xResolution}, \text{yResolution})$. Normalized device coordinate (NDC) space normalizes raster space so that x and y range from $(0, 0)$ to $(1, 1)$.

- *Camera space:* A virtual camera is placed in the scene at some world space point with a particular viewing direction and orientation. This defines a new coordinate system with its origin at the camera's location. The z axis of this coordinate system is mapped to the viewing direction, and the y axis is mapped to the up direction. This is a handy space for reasoning about which objects are potentially visible to the camera. For example, if an object's camera space bounding box is entirely behind the $z = 0$ plane (and the camera doesn't have a field of view wider than 180 degrees), the object will not be visible to the camera.
- *Screen space:* Screen space is defined on the image plane. The camera projects objects in camera space onto the image plane; the parts inside the *screen window* are visible in the image that is generated. Depth z values in screen space range from zero to one, corresponding to points at the near and far clipping planes, respectively. Note that, although this is called "screen" space, it is still a 3D coordinate system, since z values are meaningful.
- *Normalized device coordinate (NDC) space:* This is the coordinate system for the actual image being rendered. In x and y, this space ranges from $(0, 0)$ to $(1, 1)$, with $(0, 0)$ being the upper-left corner of the image. Depth values are the same as in screen space and a linear transformation converts from screen to NDC space.
- *Raster space:* This is almost the same as NDC space, except the x and y coordinates range from $(0, 0)$ to $(\text{xResolution}, \text{yResolution})$.

All cameras store a world-space-to-camera-space transformation; this can be used to transform primitives in the scene into camera space. The origin of camera space is the camera's position, and the camera points along the camera space z axis. The projective cameras in the next section use 4×4 matrices to transform between all of these spaces

Camera 302

as needed, but cameras with unusual imaging characteristics can't necessarily represent all of these transformations with matrices.

6.2 PROJECTIVE CAMERA MODELS

One of the fundamental issues in 3D computer graphics is the *3D viewing problem:* how to project a three-dimensional scene onto a two-dimensional image for display. Most of the classic approaches can be expressed by a 4×4 projective transformation matrix. Therefore, we will introduce a projection matrix camera class and then define two camera models based on it. The first implements an orthographic projection, and the other implements a perspective projection—two classic and widely used projections.

```
⟨Camera Declarations⟩ +≡
   class ProjectiveCamera : public Camera {
   public:
       ⟨ProjectiveCamera Public Methods⟩
   protected:
       ⟨ProjectiveCamera Protected Data 305⟩
   };
```

In addition to the parameters required by the `Camera` base class, the `ProjectiveCamera` takes the projective transformation matrix, screen space extent of the image, and additional parameters for depth of field. Depth of field, which will be described and implemented at the end of this section, simulates the blurriness of out-of-focus objects that occurs in real lens systems.

```
⟨Camera Method Definitions⟩ +≡
   ProjectiveCamera::ProjectiveCamera(const AnimatedTransform &cam2world,
           const Transform &proj, const float screenWindow[4], float sopen,
           float sclose, float lensr, float focald, Film *f)
       : Camera(cam2world, sopen, sclose, f) {
       ⟨Initialize depth of field parameters 314⟩
       ⟨Compute projective camera transformations 305⟩
   }
```

`ProjectiveCamera` implementations pass the projective transformation up to the base class constructor shown here. This transformation gives the camera-to-screen projection; from that, the constructor can compute most of the others that are needed.

```
⟨Compute projective camera transformations⟩ ≡ 305
   CameraToScreen = proj;
   ⟨Compute projective camera screen transformations 306⟩
   RasterToCamera = Inverse(CameraToScreen) * RasterToScreen;
```

```
⟨ProjectiveCamera Protected Data⟩ ≡ 305
   Transform CameraToScreen, RasterToCamera;
```

The only nontrivial transformation to compute in the constructor is the screen-to-raster projection. In the following code, note the composition of transformations where (reading from bottom to top), we start with a point in screen space, translate so that the

AnimatedTransform 96
Camera 302
Film 403
ProjectiveCamera 305
ProjectiveCamera::
  CameraToScreen 305
ProjectiveCamera::
  RasterToCamera 305
Transform 76

upper-left corner of the screen is at the origin, and then scale by the reciprocal of the screen width and height, giving us a point with x and y coordinates between zero and one (these are NDC coordinates). Finally, we scale by the raster resolution, so that we end up covering the entire raster range from (0, 0) up to the overall raster resolution. An important detail here is that the y coordinate is inverted by this transformation; this is necessary because increasing y values move up the image in screen coordinates, but down in raster coordinates.

⟨*Compute projective camera screen transformations*⟩ ≡ 305
```
ScreenToRaster = Scale(float(film->xResolution),
                       float(film->yResolution), 1.f) *
    Scale(1.f / (screenWindow[1] - screenWindow[0]),
          1.f / (screenWindow[2] - screenWindow[3]), 1.f) *
    Translate(Vector(-screenWindow[0], -screenWindow[3], 0.f));
RasterToScreen = Inverse(ScreenToRaster);
```

⟨*ProjectiveCamera Protected Data*⟩ +≡ 305
```
Transform ScreenToRaster, RasterToScreen;
```

6.2.1 ORTHOGRAPHIC CAMERA

⟨*OrthographicCamera Declarations*⟩ ≡
```
class OrthoCamera : public ProjectiveCamera {
public:
    ⟨OrthoCamera Public Methods⟩
private:
    ⟨OrthoCamera Private Data 307⟩
};
```

The orthographic camera, defined in the files `cameras/orthographic.h` and `cameras/orthographic.cpp`, is based on the orthographic projection transformation. The orthographic transformation takes a rectangular region of the scene and projects it onto the front face of the box that defines the region. It doesn't give the effect of *foreshortening*—objects becoming smaller on the image plane as they get farther away—but it does leave parallel lines parallel, and it preserves relative distance between objects. Figure 6.2 shows how this rectangular volume defines the visible region of the scene. Figure 6.3 compares the result of using the orthographic projection for rendering to the perspective projection defined in the next section.

The orthographic camera constructor generates the orthographic transformation matrix with the `Orthographic()` function, which will be defined shortly.

⟨*OrthographicCamera Definitions*⟩ ≡
```
OrthoCamera::OrthoCamera(const AnimatedTransform &cam2world,
        const float screenWindow[4], float sopen, float sclose,
        float lensr, float focald, Film *f)
    : ProjectiveCamera(cam2world, Orthographic(0., 1.), screenWindow,
                       sopen, sclose, lensr, focald, f) {
    ⟨Compute differential changes in origin for ortho camera rays 307⟩
}
```

AnimatedTransform 96
Film 403
Film::xResolution 403
Film::yResolution 403
OrthoCamera 306
Orthographic() 307
ProjectiveCamera 305
ProjectiveCamera::
RasterToScreen 306
ProjectiveCamera::
ScreenToRaster 306
Scale() 80
Transform 76
Translate() 79
Vector 57

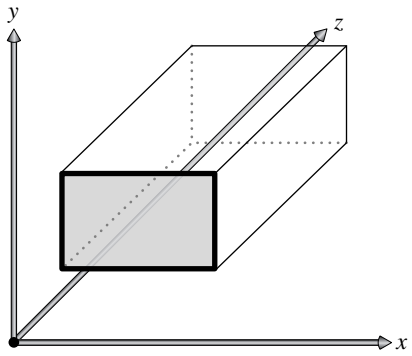


Figure 6.2: The orthographic view volume is an axis-aligned box in camera space, defined such that objects inside the region are projected onto the $z =$ hither face of the box.

The orthographic viewing transformation leaves x and y coordinates unchanged, but maps z values at the hither plane to 0 and z values at the yon plane to 1. To do this, the scene is first translated along the z axis so that the hither plane is aligned with $z = 0$. Then, the scene is scaled in z so that the yon plane maps to $z = 1$. The composition of these two transformations gives the overall transformation. (For a ray tracer like `pbrt`, we'd like the hither plane to be at 0 so that rays start at the plane that goes through the camera's position; the yon plane offset doesn't particularly matter.)

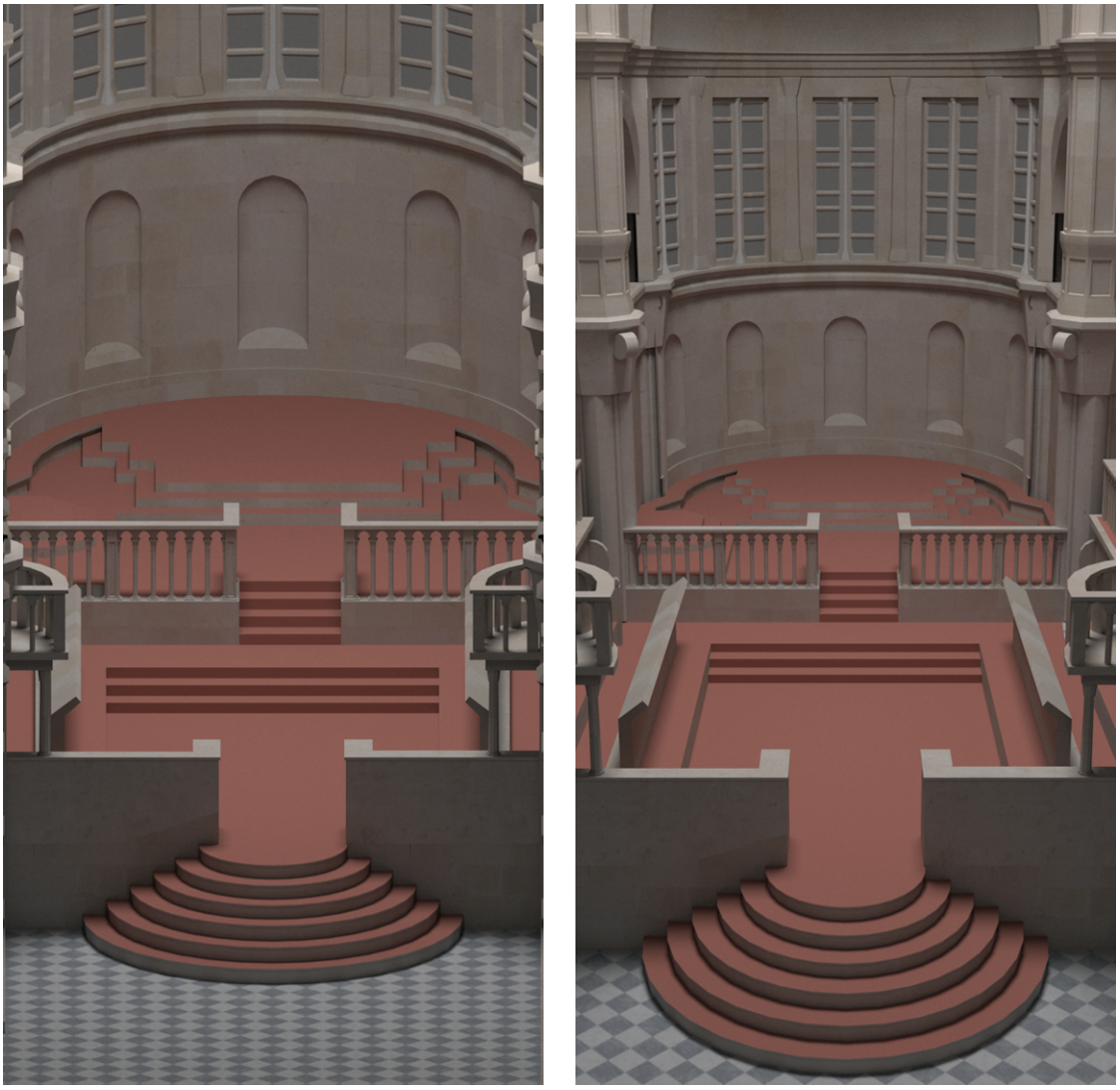
⟨*Transform Method Definitions*⟩ +≡
```
Transform Orthographic(float znear, float zfar) {
    return Scale(1.f, 1.f, 1.f / (zfar-znear)) *
           Translate(Vector(0.f, 0.f, -znear));
}
```

Thanks to the simplicity of the orthographic projection, it's easy to directly compute the differential rays in the x and y directions in the `GenerateRayDifferential()` method. The directions of the differential rays will be the same as the main ray (as they are for all rays generated by an orthographic camera), and the difference in origins will be the same for all rays. Therefore, the constructor here precomputes how much the ray origins shift in camera space coordinates due to a single pixel shift in the x and y directions on the image plane.

⟨*Compute differential changes in origin for ortho camera rays*⟩ ≡ 306
```
dxCamera = RasterToCamera(Vector(1, 0, 0));
dyCamera = RasterToCamera(Vector(0, 1, 0));
```

⟨*OrthoCamera Private Data*⟩ ≡ 306
```
Vector dxCamera, dyCamera;
```

ProjectiveCamera::
RasterToCamera 305
Scale() 80
Transform 76
Translate() 79
Vector 57



(a) (b)

Figure 6.3: Images of the Church Model. Rendered with (a) orthographic and (b) perspective cameras. Note that features like the stairs, checks on the floor, and back windows are rendered quite differently with the two models. The lack of foreshortening makes the orthographic view feel like it has less depth, although it does preserve parallel lines, which can be a useful property.

We can now go through the code to take a sample point in raster space and turn it into a camera ray. The process is summarized in Figure 6.4. First, the raster space sample position is transformed into a point in camera space, giving a point located on the hither plane, which is the origin of the camera ray. Because the camera space viewing direction points down the z axis, the camera space ray direction is (0, 0, 1).

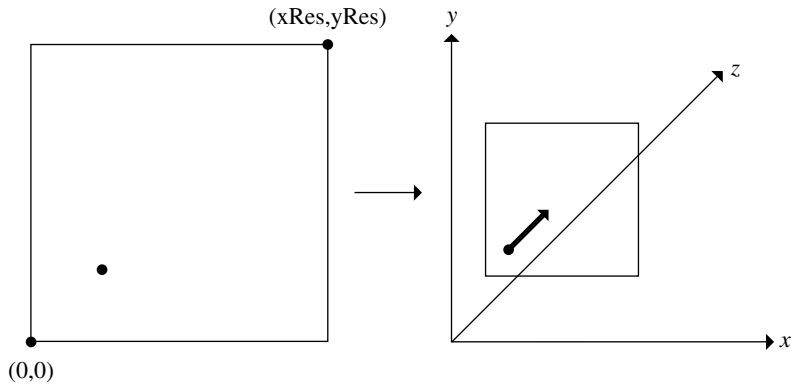


Figure 6.4: To create a ray with the orthographic camera, a raster space position on the image plane is transformed to camera space, giving the ray's origin on the hither plane. The ray's direction in camera space is $(0, 0, 1)$, down the z axis.

If depth of field has been enabled for this scene, the ray's origin and direction are modified so that depth of field is simulated. Depth of field will be explained later in this section. The ray's time value is set by linearly interpolating between the shutter open and shutter close times by the `CameraSample::time` offset (which is in the range $[0, 1)$). Finally, the ray is transformed into world space before being returned.

⟨*OrthographicCamera Definitions*⟩ +≡
```
float OrthoCamera::GenerateRay(const CameraSample &sample, Ray *ray) const {
    ⟨Generate raster and camera samples 309⟩
    *ray = Ray(Pcamera, Vector(0,0,1), 0.f, INFINITY);
    ⟨Modify ray for depth of field 315⟩
    ray->time = Lerp(sample.time, shutterOpen, shutterClose);
    CameraToWorld(*ray, ray);
    return 1.f;
}
```

Camera::CameraToWorld 302
Camera::shutterClose 302
Camera::shutterOpen 302
CameraSample 342
CameraSample::imageX 342
CameraSample::imageY 342
CameraSample::time 342
INFINITY 1002
Lerp() 1000
OrthoCamera 306
Point 63
ProjectiveCamera::
RasterToCamera 305
Ray 66
Vector 57

Once all of the transformation matrices have been set up, it's easy to set up the raster space sample point and transform it to camera space.

⟨*Generate raster and camera samples*⟩ ≡ 309, 312
```
Point Pras(sample.imageX, sample.imageY, 0);
Point Pcamera;
RasterToCamera(Pras, &Pcamera);
```

The implementation of `GenerateRayDifferential()` performs the same computation to generate the main camera ray. The differential ray origins are found using the offsets computed in the `OrthoCamera` constructor and then the full ray differential is transformed to world space.

⟨*OrthographicCamera Definitions*⟩ +≡
```
float OrthoCamera::GenerateRayDifferential(const CameraSample &sample,
        RayDifferential *ray) const {
    ⟨Compute main orthographic viewing ray⟩
    ray->rxOrigin = ray->o + dxCamera;
    ray->ryOrigin = ray->o + dyCamera;
    ray->rxDirection = ray->ryDirection = ray->d;
    ray->hasDifferentials = true;
    CameraToWorld(*ray, ray);
    return 1.f;
}
```

6.2.2 PERSPECTIVE CAMERA

The perspective projection is similar to the orthographic projection in that it projects a volume of space onto a 2D image plane. However, it includes the effect of foreshortening: objects that are far away are projected to be smaller than objects of the same size that are closer. Unlike the orthographic projection, the perspective projection doesn't preserve distances or angles, and parallel lines no longer remain parallel. The perspective projection is a reasonably close match to how an eye or camera lens generates images of the three-dimensional world. The perspective camera is implemented in the files `cameras/perspective.h` and `cameras/perspective.cpp`.

⟨*PerspectiveCamera Declarations*⟩ ≡
```
class PerspectiveCamera : public ProjectiveCamera {
public:
    ⟨PerspectiveCamera Public Methods 312⟩
private:
    ⟨PerspectiveCamera Private Data 312⟩
};
```

⟨*PerspectiveCamera Method Definitions*⟩ ≡
```
PerspectiveCamera:: PerspectiveCamera(const AnimatedTransform &cam2world,
        const float screenWindow[4], float sopen, float sclose,
        float lensr, float focald, float fov, Film *f)
    : ProjectiveCamera(cam2world, Perspective(fov, 1e-2f, 1000.f),
                       screenWindow, sopen, sclose, lensr, focald, f) {
    ⟨Compute differential changes in origin for perspective camera rays 312⟩
}
```

The perspective projection describes perspective viewing of the scene. Points in the scene are projected onto a viewing plane perpendicular to the z axis. The `Perspective()` function computes this transformation; it takes a field-of-view angle in `fov`, and the distances to a near z plane and a far z plane. After the perspective projection, points at the near z plane are mapped to have $z = 0$ and points at the far plane have $z = 1$. (Figure 6.5). For rendering systems based on rasterization, it's important to set the positions of these planes carefully; they determine the z range of the scene that is rendered, but setting them with too many orders of magnitude variation between their values can lead to numeric precision errors. For a ray tracers like `pbrt`, they can be set arbitrarily as they are here.

AnimatedTransform 96
Camera::CameraToWorld 302
CameraSample 342
Film 403
OrthoCamera 306
OrthoCamera::dxCamera 307
OrthoCamera::dyCamera 307
Perspective() 311
PerspectiveCamera 310
ProjectiveCamera 305
Ray::d 67
Ray::o 67
RayDifferential 69
RayDifferential::
hasDifferentials 69
RayDifferential::
rxDirection 69
RayDifferential::rxOrigin 69
RayDifferential::
ryDirection 69
RayDifferential::ryOrigin 69

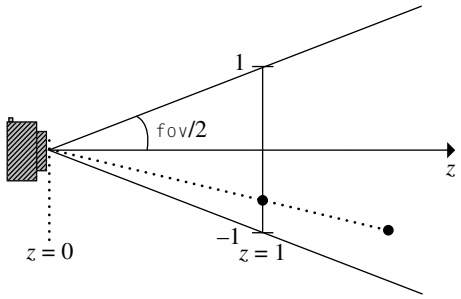


Figure 6.5: The perspective transformation matrix projects points in camera space onto the image plane. The x' and y' coordinates of the projected points are equal to the unprojected x and y coordinates divided by the z coordinate. The projected z' coordinate is computed so that points on the near plane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

⟨*Transform Method Definitions*⟩ +≡
```
Transform Perspective(float fov, float n, float f) {
    ⟨Perform projective divide 311⟩
    ⟨Scale to canonical viewing volume 312⟩
}
```

The transformation is most easily understood in two steps:

1. Points p in camera space are projected onto the viewing plane. A bit of algebra shows that the projected x' and y' coordinates on the viewing plane can be computed by dividing x and y by the point's z coordinate value. The projected z depth is remapped so that z values at the hither plane are 0 and z values at the yon plane are 1. The computation we'd like to do is

$$x' = x/z$$
$$y' = y/z$$
$$z' = \frac{f(z-n)}{z(f-n)}.$$

   All of this computation can be encoded in a 4×4 matrix using homogeneous coordinates:

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

⟨*Perform projective divide*⟩ ≡ 311
```
Matrix4x4 persp = Matrix4x4(1, 0,          0,              0,
                            0, 1,          0,              0,
                            0, 0, f / (f - n), -f*n / (f - n),
                            0, 0,          1,              0);
```
Matrix4x4 1021
Transform 76

2. The angular field of view (`fov`) specified by the user is accounted for by scaling the (x, y) values on the projection plane so that points inside the field of view project to coordinates between $[-1, 1]$ on the view plane. For square images, both x and y lie between $[-1, 1]$ in screen space. Otherwise, the direction in which the image is narrower maps to $[-1, 1]$ and the wider direction maps to a proportionally larger range of screen space values. Recall that the tangent is equal to the ratio of the opposite side of a right triangle to the adjacent side. Here the adjacent side has length 1, so the opposite side has the length $\tan(\texttt{fov}/2)$. Scaling by the reciprocal of this length maps the field of view to range from $[-1, 1]$.

⟨*Scale to canonical viewing volume*⟩ ≡ 311
```
float invTanAng = 1.f / tanf(Radians(fov) / 2.f);
return Scale(invTanAng, invTanAng, 1) * Transform(persp);
```

Similar to the `OrthoCamera`, information about how the camera rays generated by the `PerspectiveCamera` change as we shift pixels on the image plane can be precomputed in the constructor. Here, we compute the change in position on the near perspective plane in camera space with respect to shifts in pixel location.

⟨*Compute differential changes in origin for perspective camera rays*⟩ ≡ 310
```
dxCamera = RasterToCamera(Point(1,0,0)) - RasterToCamera(Point(0,0,0));
dyCamera = RasterToCamera(Point(0,1,0)) - RasterToCamera(Point(0,0,0));
```

⟨*PerspectiveCamera Private Data*⟩ ≡ 310
```
Vector dxCamera, dyCamera;
```

With the perspective projection, all rays originate from the origin, (0, 0, 0), in camera space. A ray's direction is given by the vector from the origin to the point on the near plane `Pcamera` that corresponds to the provided `CameraSample`. In other words, the ray's vector direction is componentwise equal to this point's position, so rather than doing a useless subtraction to compute the direction, we just initialize the direction directly from the point `Pcamera`.

⟨*PerspectiveCamera Method Definitions*⟩ +≡
```
float PerspectiveCamera::GenerateRay(const CameraSample &sample,
                                     Ray *ray) const {
    ⟨Generate raster and camera samples 309⟩
    *ray = Ray(Point(0,0,0), Vector(Pcamera), 0.f, INFINITY);
    ⟨Modify ray for depth of field 315⟩
    ray->time = Lerp(sample.time, shutterOpen, shutterClose);
    CameraToWorld(*ray, ray);
    return 1.f;
}
```

The `GenerateRayDifferential()` method follows the implementation of `GenerateRay()`, except for an additional fragment that computes the differential rays.

⟨*PerspectiveCamera Public Methods*⟩ ≡ 310
```
float GenerateRayDifferential(const CameraSample &sample,
                              RayDifferential *ray) const;
```

Camera::CameraToWorld 302
Camera::shutterClose 302
Camera::shutterOpen 302
CameraSample 342
CameraSample::time 342
INFINITY 1002
Lerp() 1000
OrthoCamera 306
PerspectiveCamera 310
PerspectiveCamera::dxCamera 312
PerspectiveCamera::dyCamera 312
Point 63
ProjectiveCamera::RasterToCamera 305
Radians() 1001
Ray 66
Ray::time 67
RayDifferential 69
Scale() 80
Transform 76
Vector 57

⟨*Compute offset rays for* PerspectiveCamera *ray differentials*⟩ ≡

```
ray->rxOrigin = ray->ryOrigin = ray->o;
ray->rxDirection = Normalize(Vector(Pcamera) + dxCamera);
ray->ryDirection = Normalize(Vector(Pcamera) + dyCamera);
```

6.2.3 DEPTH OF FIELD

Real cameras have lens systems that focus light through a finite-sized aperture onto the film plane. Because the aperture has finite area, a single point in the scene may be projected onto an area on the film plane called the *circle of confusion* (Figure 6.6). Correspondingly, a finite area of the scene may be visible from a single point on the image plane, giving a blurred image. Figures 6.7 and 6.8 show this effect, depth of field, in a scene with a series of copies of the dragon model. Figure 6.7(a) is rendered with an infinitesimal aperture and thus without any depth of field effects. Figures 6.7(b) and 6.8 show the increase in blurriness as the size of the lens aperture is increased. Note that the second dragon from the right remains in focus throughout all of the images, as the plane of focus has been placed at its depth. Figure 6.9 shows depth of field used to render the ecosystem scene. Note how the effect draws the viewer's eye to the in-focus tree in the center of the image.

The size of the circle of confusion is affected by the radius of the aperture and the distance between the object and the lens. The *focal distance* is the distance from the lens to the plane of objects that project to a zero-radius circle of confusion. These points appear to be perfectly in focus. In practice, objects do not have to be exactly on the focal plane to appear in sharp focus; as long as the circle of confusion is roughly smaller than a pixel,

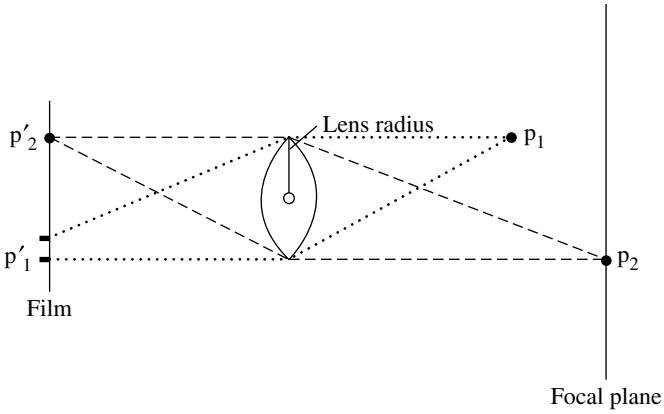
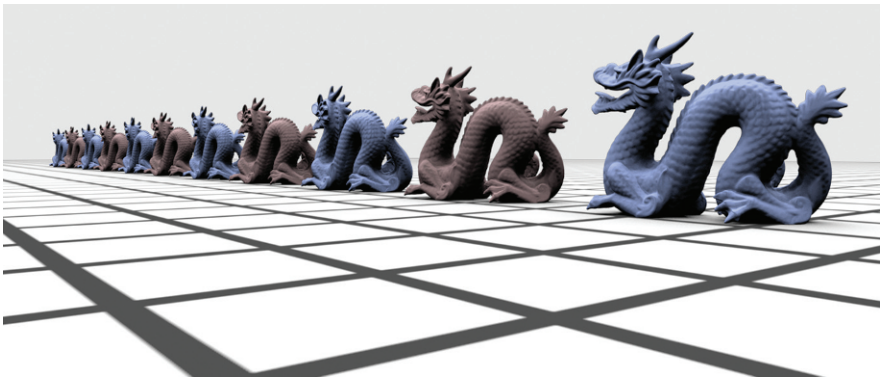
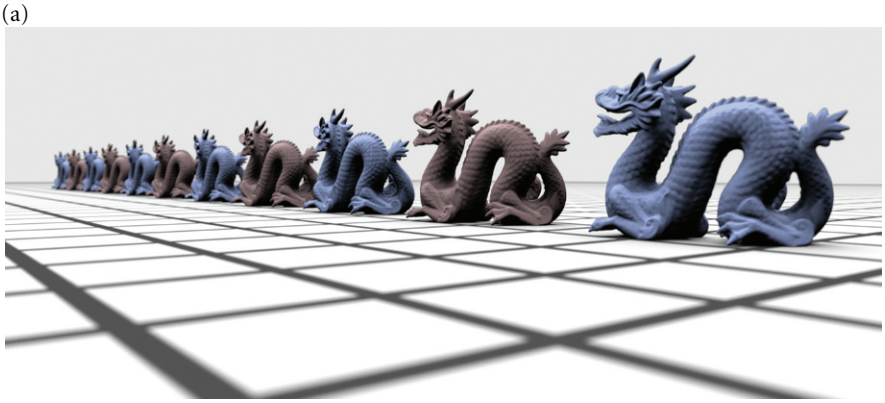


Figure 6.6: Real-world cameras have a lens with finite aperture and lens controls that adjust the lens position with respect to the film plane. Because the aperture is finite, objects in the scene aren't all imaged onto the film in perfect focus. Here, the point p_1 doesn't lie on the plane of points in perfect focus, so it images to an area p'_1 on the film and is blurred. The point p_2 does lie on the focal plane, so it images to a point p'_2 and is in focus. Either increasing the aperture size or increasing an object's distance from the focal plane will increase an out-of-focus object's blurriness.

PerspectiveCamera 310
PerspectiveCamera::
dxCamera 312
PerspectiveCamera::
dyCamera 312
RayDifferential::
rxDirection 69
RayDifferential::rxOrigin 69
RayDifferential::
ryDirection 69
RayDifferential::ryOrigin 69
Vector 57



(a)



(b)

Figure 6.7: (a) Scene rendered with no depth of field and (b) depth of field due to a relatively small lens aperture, which gives only a small amount of blurriness in the out-of-focus regions.

objects appear to be in focus. The range of distances from the lens at which objects appear in focus is called the lens's *depth of field*.

Projective cameras take two extra parameters for depth of field: one sets the size of the lens aperture, and the other sets the focal distance.

⟨*ProjectiveCamera Protected Data*⟩ +≡ 305
```
float lensRadius, focalDistance;
```

⟨*Initialize depth of field parameters*⟩ ≡ 305
```
lensRadius = lensr;
focalDistance = focald;
```

The math behind computing circles of confusion for simple lenses is not difficult; it mostly involves repeated application of similar triangles and some reasonable approximations about the shape of the lens profile. This process can be easily modeled in a ray tracer with just a few lines of code. All that is necessary is to choose a point on the lens and find the appropriate ray that starts on the lens at that point such that objects in the plane

ProjectiveCamera::
focalDistance 314

ProjectiveCamera::
lensRadius 314

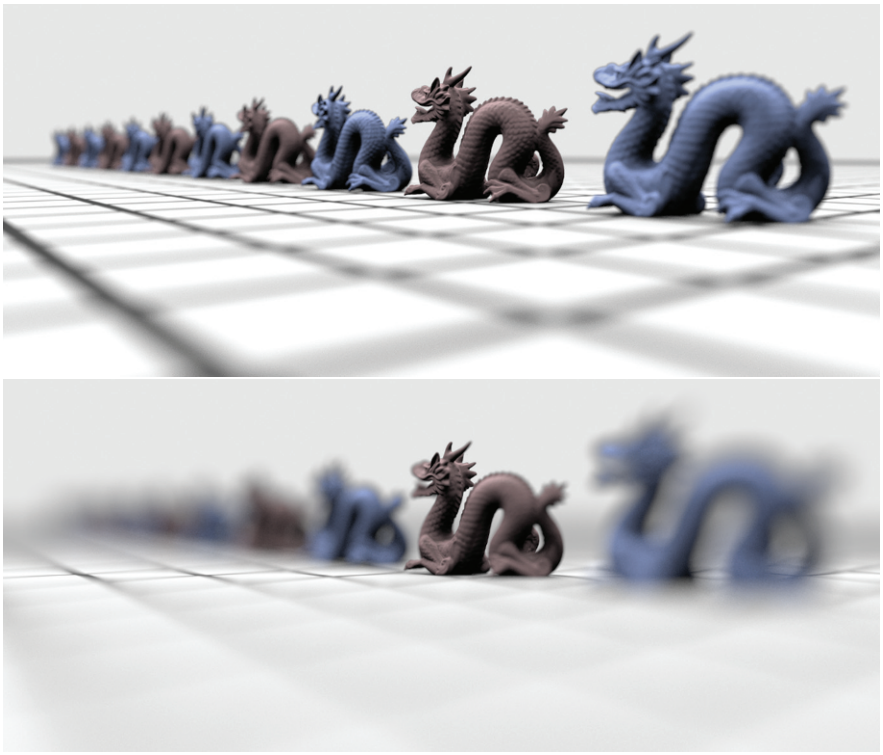


Figure 6.8: As the size of the lens aperture increases, the size of the circle of confusion in the out-of-focus areas increases, giving a greater amount of blur on the image plane.

of focus are in focus (Figure 6.10). It is generally necessary to trace many rays for each image pixel in order to adequately sample the lens for smooth depth of field. Figure 6.11 shows the ecosystem scene from Figure 6.9 with only four samples per pixel (Figure 6.9 had 128 samples per pixel).

⟨*Modify ray for depth of field*⟩ ≡ **309, 312**

```
if (lensRadius > 0.) {
    ⟨Sample point on lens 317⟩
    ⟨Compute point on plane of focus 318⟩
    ⟨Update ray for effect of lens 318⟩
}
```

CameraSample 342
CameraSample::lensU 342
CameraSample::lensV 342
ConcentricSampleDisk() 667
ProjectiveCamera::lensRadius 314

The `ConcentricSampleDisk()` function, defined in Chapter 13, takes a (u, v) sample position in $[0, 1]^2$ and maps it to a 2D unit disk centered at the origin (0, 0). To turn this into a point on the lens, these coordinates are scaled by the lens radius. The `CameraSample` class provides the (u, v) lens-sampling parameters in the `CameraSample::lensU` and `CameraSample::lensV` fields.

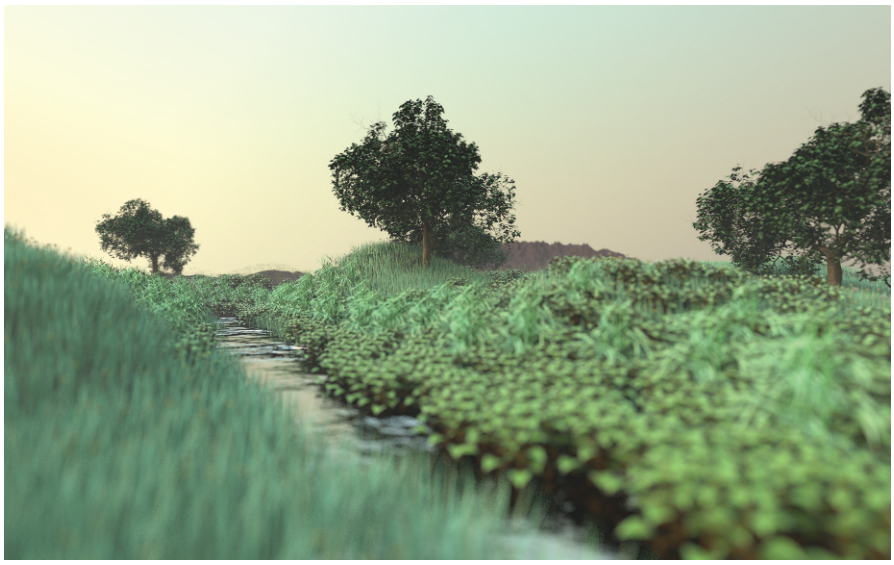


Figure 6.9: Depth of field gives a greater sense of depth and scale to the ecosystem scene.

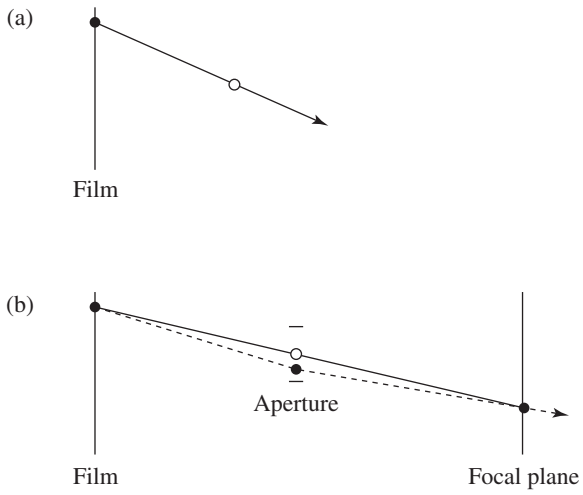


Figure 6.10: (a) For a pinhole camera model, a single camera ray is associated with each point on the film plane (filled circle), given by the ray that passes through the single point of the pinhole lens (empty circle). (b) For a camera model with a finite aperture, we sample a point (filled circle) on the disk-shaped lens for each ray. We can also compute the ray that passes through the center of the lens (corresponding to the pinhole model) and the point where it intersects the focal plane (solid line). We know that all objects in the focal plane 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 focal plane.

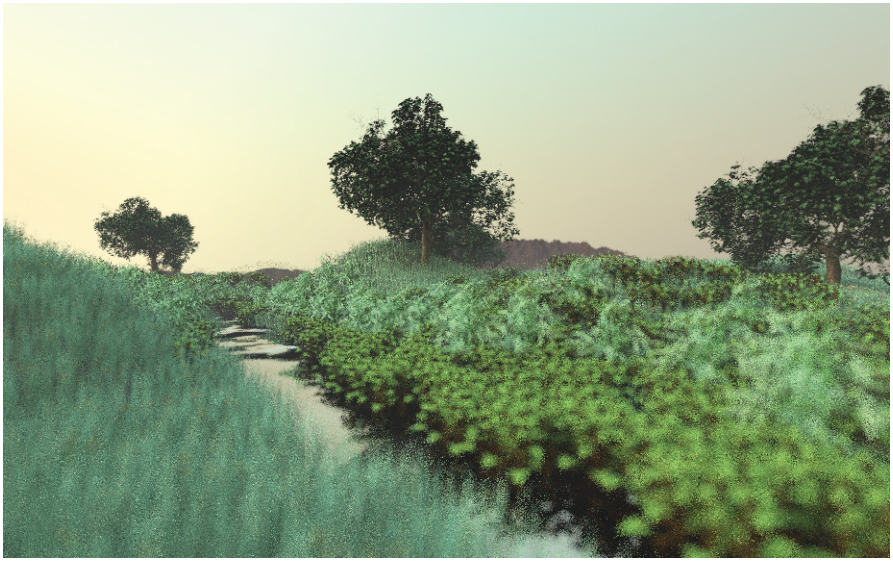


Figure 6.11: Ecosytem scene with depth of field and only four samples per pixel: the depth of field is undersampled and the image is grainy.

⟨*Sample point on lens*⟩ ≡ 315

```
float lensU, lensV;
ConcentricSampleDisk(sample.lensU, sample.lensV, &lensU, &lensV);
lensU *= lensRadius;
lensV *= lensRadius;
```

The ray's origin is this point on the lens. Now it is necessary to determine the proper direction for the new ray. This direction could be computed using Snell's law, which describes how light refracts when passing from one medium (e.g., air) to another (e.g., glass), but the specifics of this particular problem make this simpler.[1] We know that *all* rays from the given image sample through the lens must converge at the same point on the focal plane. Furthermore, we know that rays through the center of the lens are not refracted, so finding the appropriate point of convergence is a matter of intersecting the unperturbed ray from the pinhole model with the focal plane and then setting the ray's direction to be the vector from the point on the lens to the intersection point.

For this simple model, the focal plane is perpendicular to the z axis and the ray starts at the origin, so intersecting the ray through the lens center with the focal plane is straightforward. The t value of the intersection is given by

$$t = \frac{\text{focalDistance}}{\mathbf{d}_z}.$$

CameraSample::lensU 342
CameraSample::lensV 342
ConcentricSampleDisk() 667
ProjectiveCamera::
lensRadius 314

1 We are also assuming a single thin spherical lens element. Simulating complex lens systems with multiple glass elements is much more involved; see the exercises and the "Further Reading" section.

⟨*Compute point on plane of focus*⟩ ≡ 315
```
float ft = focalDistance / ray->d.z;
Point Pfocus = (*ray)(ft);
```

Now the ray can be initialized. The origin is set to the sampled point on the lens and the direction is set so that the ray passes through the point on the plane of focus, `Pfocus`.

⟨*Update ray for effect of lens*⟩ ≡ 315
```
ray->o = Point(lensU, lensV, 0.f);
ray->d = Normalize(Pfocus - ray->o);
```

6.3 ENVIRONMENT CAMERA

One advantage of ray tracing compared to scan line or rasterization-based rendering methods is that it's easy to employ unusual image projections. We have great freedom in how the image sample positions are mapped into ray directions, since the rendering algorithm doesn't depend on properties such as straight lines in the scene always projecting to straight lines in the image.

In this section, we will describe a camera model that traces rays in all directions around a point in the scene, giving a two-dimensional view of everything that is visible from that point. Consider a sphere around the camera position in the scene; choosing points on that sphere gives directions to trace rays in. If we parameterize the sphere with spherical coordinates, each point on the sphere is associated with a (θ, ϕ) pair, where $\theta \in [0, \pi]$ and $\phi \in [0, 2\pi]$. (See Section 5.5.2 for more details on spherical coordinates.) This type of image is particularly useful because it represents all of the incident light at a point on the scene. It will be useful later when we discuss environment lighting—a rendering technique that uses image-based representations of light in a scene. Figure 6.12 shows this camera in action with the San Miguel model. θ values range from 0 at the top of the image to π at the bottom of the image, and ϕ values range from 0 to 2π, moving from left to right across the image.[2]

⟨*EnvironmentCamera Declarations*⟩ ≡
```
class EnvironmentCamera : public Camera {
public:
    ⟨EnvironmentCamera Public Methods 319⟩
};
```

The `EnvironmentCamera` derives directly from the `Camera` class, not the `ProjectiveCamera` class. This is because the environmental projection is nonlinear and cannot be captured by a single 4×4 matrix. It is defined in the files `cameras/environment.h` and `cameras/environment.cpp`.

Camera 302
EnvironmentCamera 318
Point 63
ProjectiveCamera 305
ProjectiveCamera::
focalDistance 314
Ray::d 67
Ray::o 67
Vector::Normalize() 63

2 Readers familiar with cartography will recognize this as an equirectangular projection.

Figure 6.12: 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 15.

⟨*EnvironmentCamera Public Methods*⟩ ≡ 318
```
EnvironmentCamera(const AnimatedTransform &cam2world, float sopen,
                  float sclose, Film *film)
    : Camera(cam2world, sopen, sclose, film) {
}
```

⟨*EnvironmentCamera Method Definitions*⟩ ≡
```
float EnvironmentCamera::GenerateRay(const CameraSample &sample,
                                     Ray *ray) const {
    float time = Lerp(sample.time, shutterOpen, shutterClose);
    ⟨Compute environment camera ray direction 319⟩
    *ray = Ray(Point(0,0,0), dir, 0.f, INFINITY, time);
    CameraToWorld(*ray, ray);
    return 1.f;
}
```

AnimatedTransform 96
Camera 302
Camera::CameraToWorld 302
Camera::shutterClose 302
Camera::shutterOpen 302
CameraSample 342
CameraSample::imageX 342
CameraSample::imageY 342
CameraSample::time 342
EnvironmentCamera 318
Film 403
Film::xResolution 403
Film::yResolution 403
INFINITY 1002
Lerp() 1000
M_PI 1002
Point 63
Ray 66
Vector 57

To compute the (θ, ϕ) coordinates for this ray, NDC coordinates are computed from the raster image sample position and then scaled to cover the (θ, ϕ) range. Next, the spherical coordinate formula is used to compute the ray direction, and finally the direction is converted to world space. (Note that because the y direction is "up" in camera space, here the x and y coordinates in the spherical coordinate formula are exchanged in comparison to usage elsewhere in the system.)

⟨*Compute environment camera ray direction*⟩ ≡ 319
```
float theta = M_PI * sample.imageY / film->yResolution;
float phi = 2 * M_PI * sample.imageX / film->xResolution;
Vector dir(sinf(theta) * cosf(phi), cosf(theta),
           sinf(theta) * sinf(phi));
```

FURTHER READING

In his seminal Sketchpad system, Sutherland (1963) was the first to use projection matrices for computer graphics. Akenine-Möller, Haines, and Hoffman (2008) have provided a particularly well-written derivation of the orthographic and perspective projection matrices in *Real Time Rendering*. Other good references for projections are Rogers and Adams's *Mathematical Elements for Computer Graphics* (1990), Watt and Watt (1992), Foley et al. (1990), and Eberly's book on game engine design (Eberly 2001).

Potmesil and Chakravarty (1981, 1982, 1983) did early work on depth of field and motion blur in computer graphics. Cook and collaborators developed a more accurate model for these effects based on *distribution ray tracing*; this is the approach used for the depth of field calculations in this chapter (Cook, Porter, and Carpenter 1984; Cook 1986).

Kolb, Mitchell, and Hanrahan (1995) investigated simulating complex camera lens systems with ray tracing in order to model the imaging effects of real cameras. Another unusual projection method was used by Greene and Heckbert (1986) for generating images for Omnimax® theaters. The `EnvironmentCamera` in this chapter is similar to the camera model described by Musgrave (1992).

EXERCISES

6.1 Modify the way that time samples are generated in the camera implementations in this chapter to support different shutter models. For example, many cameras expose the film by sliding a rectangular slit across the film. This leads to interesting effects when objects are moving in a different direction than the exposure slit (Glassner 1999; Stephenson 2006). Implement this slit shutter model for both horizontal and vertical slits, and also a leaf shutter. Create scenes that clearly show the effects of each shutter type.

6.2 The standard model for depth of field in computer graphics models the circle of confusion as imaging a point in the scene to a circle with uniform intensity, although many real lenses produce circles of confusion with nonlinear variation such as a Gaussian distribution. This effect is known as "Bokeh" (Buhler and Wexler 2002). For example, catadioptric (mirror) lenses produce doughnut-shaped highlights when small points of light are viewed out of focus. Modify the implementation of depth of field in `pbrt` to produce images with this effect (for example, by biasing the distribution of lens sample positions). Render images showing the difference between this and the standard model.

6.3 Write an application that loads images rendered by the `EnvironmentCamera` and uses texture mapping to apply them to a sphere centered at the eyepoint such that they can be viewed interactively. The user should be able to freely change the viewing direction. If the appropriate texture-mapping function is used for generating texture coordinates on the sphere, the image generated by the application will appear as if the viewer was at the camera's location in the scene when it was rendered, thus giving the user the ability to interactively look around the scene.

`EnvironmentCamera` 318

● 6.4 Kolb, Mitchell, and Hanrahan (1995) have described a camera model for ray tracing based on simulating the lens system of a real camera, which is comprised of a set of glass lenses arranged to form an image on the film plane. Read their paper and implement a camera model in `pbrt` that implements their algorithm for following rays through lens systems. Test your implementation with some of the lens description data from their paper.