CHAPTER 31

TILT-SHIFT RENDERING USING A THIN LENS MODEL

Andrew Kensler
Amazon, formerly Pixar Animation Studios

ABSTRACT

A tilt-shift lens is a lens whose longitudinal axis can be moved out of alignment with the center of the film or image sensor—either by rotation for tilt or translation for shift. In this chapter, we will review the classic thin lens camera model in ray tracing, which sits at the sweet spot between the simple pinhole model and the complex lens systems. Then we will show how to extend it to model a tilt-shift lens. We will also show how to solve for the placement of the lens to achieve an arbitrary plane of focus.

31.1 INTRODUCTION

Tilted lenses frequently have shallow apparent depths of field, which make them popular for creating a miniaturized look (e.g., Figure 31-1a). Though it is common to fake this look with Gaussian blurs masked along a gradient, the real effect can be subtle and much more interesting. See https://www.reddit.com/r/tiltshift/ for many examples, both real and faked.

(a) *Tilted lens* **(b)** *Tilted and shifted lens*

Figure 31-1. *An example scene. (a) Miniaturized appearance with tilted lens. (b) Combining lens tilt and shift. Lens shift is used so that the center of perspective is in the upper left, and lens tilt has been applied so that the plane of focus is perpendicular to the camera and aligned to emphasize the building fronts on the right. Unlike with a standard lens, all of the fronts are in sharp focus regardless of distance down the street.*

A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_31

On the other hand, lenses that can be shifted off-center can be used to correct for perspective distortion (also known as the keystoning effect), and to switch between one-, two-, and three-point perspective.

Commonly, these abilities are combined in a tilt-shift lens (e.g., Figure 31-1b). Early examples of these could be found in the view camera where the lens and the film are held in place at opposite ends of a flexible bellows. For details, see Merklinger [2]. For an overview of photography, refer back to Chapter 1.

In this chapter, Section 31.2 will go over the traditional thin lens model in rendering, first introduced to computer graphics by Potmesil and Chakravarty [3] and to ray tracing by Cook et al. [1]. Then, Section 31.3 will cover how to add lens shift to it, while Section 31.4 will address how to modify it for tilted lenses. Since direct control is surprisingly finicky, Section 31.5 will address how to solve for the correct lens and sensor positions to achieve an arbitrary plane of focus. Finally, Section 31.6 will show some more examples of tilted lens rendering and some interesting consequences.

31.2 THIN LENS MODEL

For this chapter, we will assume that everything is in camera space, with $+\hat{\mathbf{z}}$ pointing down the view direction and the lens centered on the origin. The sample code is written for a left-handed Y-up camera coordinate system. Some distance ahead of the lens is an object at a point P that we would like to image to a point P' on the sensor behind the lens. There is an aperture that restricts the size of the cone of light passing through the lens. See Figure 31-2.

The focal length of the lens, f, and the perpendicular distances between points on the lens, object, and image planes are all related by Gauss's thin lens equation, where p is the distance from the lens to the object and p' is the distance from the lens to the image plane:

$$\frac{1}{f} = \frac{1}{p} + \frac{1}{p'}. \tag{31.1}$$

We can also reformulate the thin lens equation to compute one distance if we know the other. In this case, we can determine how far back from the lens a given point will focus to. The sensor must be placed here in order for the point to be in focus:

$$p' = \frac{pf}{p - f}. \tag{31.2}$$

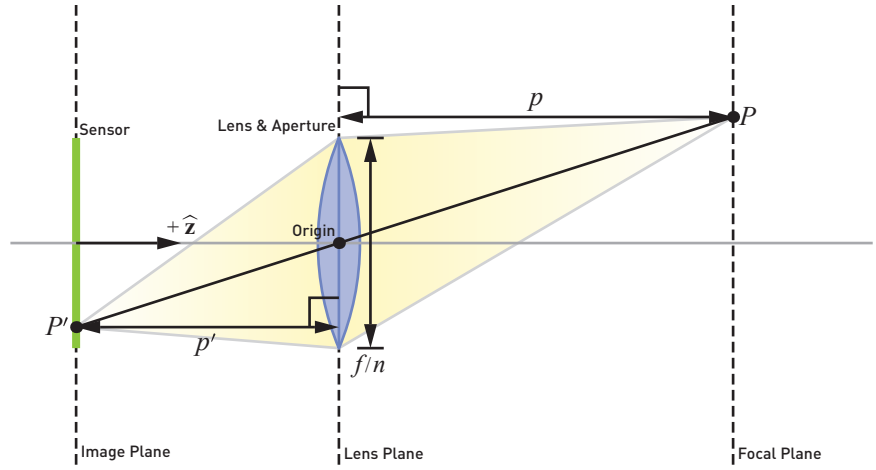


Figure 31-2. *The geometry of the thin lens model.*

Note that we must be very careful with signs here: for a lens projecting the object to a real image such as this, the focal length and two distances are positive, while the z-coordinates of points are negative when they are behind the lens.

If an object point is off of the focal plane, then it will focus to a point off of the image plane. With a circular aperture, this creates an image of the aperture on the sensor called the *circle of confusion*. The size of the acceptable circle of confusion determines the near and far limits around the focal plane for what is considered in-focus. The distance between these two limits defines the depth of field. See Figure 31-3. These near and far limits also form planes. Note that with a standard lens setup, the image plane, lens plane, focal plane, and near and far focal planes are all parallel.

One further assumption of the thin lens model is that light between an object and its image passes through the center of a thin lens undeflected. Consequently, the coordinates of the two points in camera space are related by a simple scaling factor.

To render with this, we first choose a desired focal distance for where we want the focal plane to be and use it as p in Equation 31.2. Define the distance between the lens center and the image plane with the sensor as the result of that equation, $s = p'$. Then for each pixel in an image, we simply take the location of the pixel on our sensor, P' (where $P'_z = -s$), and map it to the point

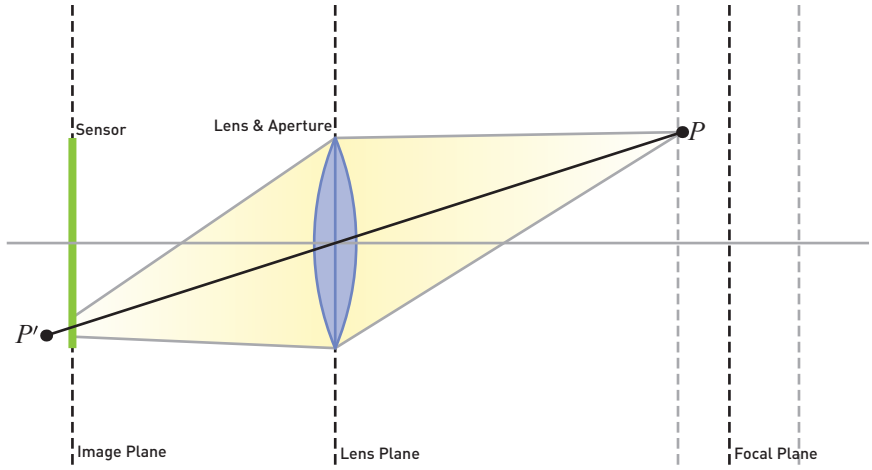


Figure 31-3. *Defocus and the circle of confusion. The object projects to a point behind the sensor and so a small circle of light from it impinges on the sensor.*

on the focal plane, P, that focuses there (note that the scaling factor relating them is constant):

$$P = P' \frac{f}{f - s}. \tag{31.3}$$

Then we aim a ray toward that point, in direction $\widehat{\mathbf{d}}$ from a randomly sampled point O within the lens aperture (Figure 31-4).

The following GLSL ES (WebGL) sample code puts this all together. It takes as input a screen coordinate and a pair of random numbers. The screen coordinate is the pixel position normalized so that the short edge lies in the range [–1, 1]. The sensor size is then the size of this short edge. This makes the field of view angle implicit on the sensor size, the focal distance, and the focal length. Though we could specify a desired field of view and then derive the sensor size to use instead, physical cameras have constant sensor sizes and so adjusting the focal distance changes the field of view angle. This is sometimes called *focus breathing* and is modeled here.

The random numbers given should be uniformly distributed within the $[0, 1]^2$ square. Ideally, these would be generated using quasi-Monte Carlo for quicker convergence. Note that in the sample code here we use them to uniformly sample a circular disk for the lens position, which yields perfectly flat, circular bokeh blurs. Alternately, we could use them to sample a polygon, sample with an uneven distribution, or even sample a density image with

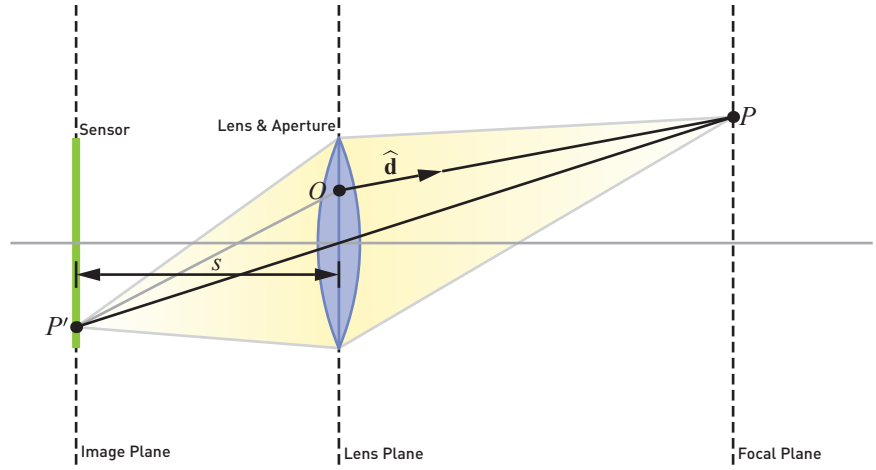


Figure 31-4. *Sampling the lens with a ray. The ray is traced from a random position on the lens to where the pixel on the sensor projects onto the focal plane.*

rounded iris blades, dust, and scratches. The choice of distributions will directly determine the appearance of the bokeh.

```
void thin_lens(vec2 screen, vec2 random,
               out vec3 ray_origin, out vec3 ray_direction)
{
    // f  : focal_length      p : focal_distance
    // n  : f_stop            P : focused
    // s  : image_plane       O : ray_origin
    // P' : sensor            d : ray_direction

    // Lens values (precomputable)
    float aperture = focal_length / f_stop;
    // Image plane values (precomputable)
    float image_plane = focal_distance * focal_length /
        (focal_distance - focal_length);

    // Image plane values (render-time)
    vec3 sensor = vec3(screen * 0.5 * sensor_size, -image_plane);
    // Lens values (render-time)
    float theta = 6.28318531 * random.x;
    float r = aperture * sqrt(random.y);
    vec3 lens = vec3(cos(theta) * r, sin(theta) * r, 0.0);
    // Focal plane values (render-time)
    vec3 focused = sensor * focal_length /
        (focal_length - image_plane);

    ray_origin = lens;
    ray_direction = normalize(focused - lens);
}
```

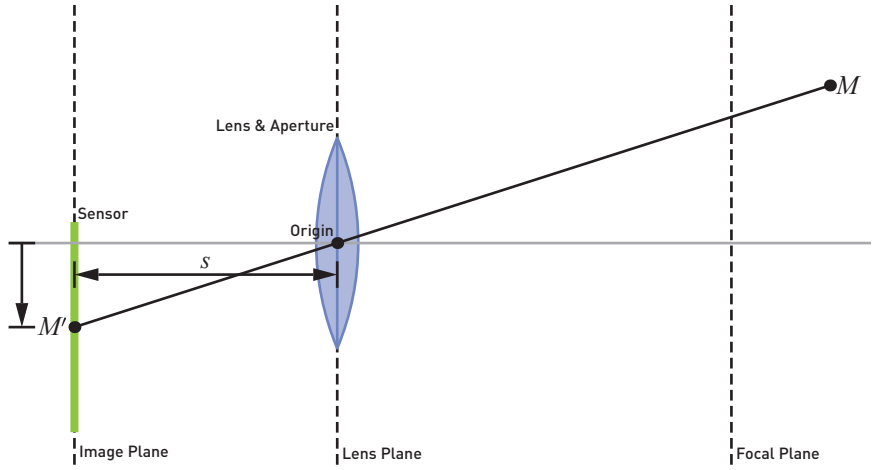


Figure 31-5. *Translating the sensor for lens shift. The point M that is shifted to appear in the image center does not need to be on the focal plane.*

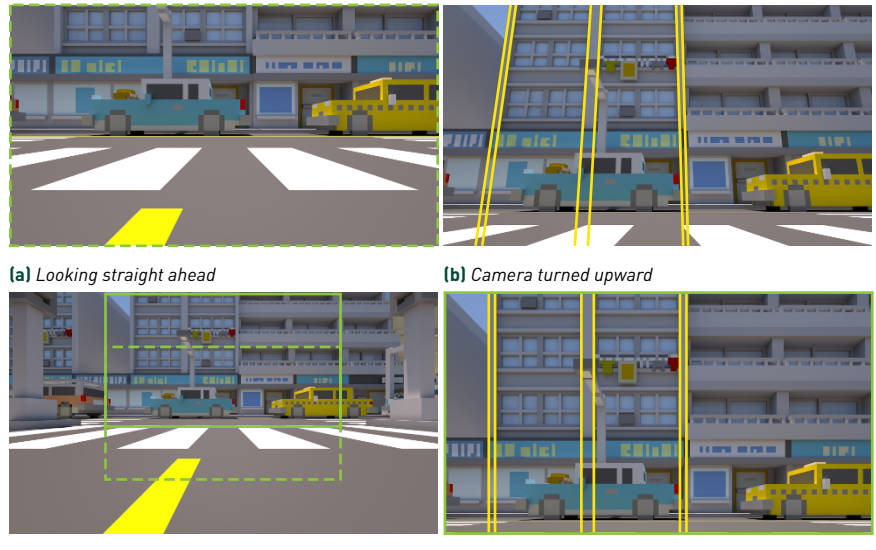
31.3 LENS SHIFT

Now consider adding lens shift to this model. Since this means that the center of the sensor is no longer in alignment with the center of the lens, we could model this as a translation of the lens within the lens plane. However, it is easier to keep the lens centered at the origin in camera space and instead translate the sensor within the image plane (Figure 31-5). This corresponds to a simple addition to the x- and y-coordinates (but not the z-coordinate) of P' for a pixel on the sensor before we map it to P on the focal plane and project a ray through it as before.

If there is a point M that we would like to shift to be centered in the middle of the image, then we can project it through the center of the lens and onto the image plane at M'. Then $(M'_x, M'_y, 0)$ gives the translation needed:

$$M' = M\frac{-s}{M_z}. \tag{31.4}$$

Commonly, this effect is used for architectural photography. As an example, suppose that we have a picture near street level and we would like to look at things a little higher up. When looking straight ahead level (Figure 31-6a), the upright lines on the vertical elements are parallel. However, if we just swivel or turn the camera upward, this creates a foreshortening and causes the lines on the upright elements to converge rather than remain parallel as before



(a) *Looking straight ahead*

(b) *Camera turned upward*

(c) *Larger image with wider angle*

(d) *Lens shift*

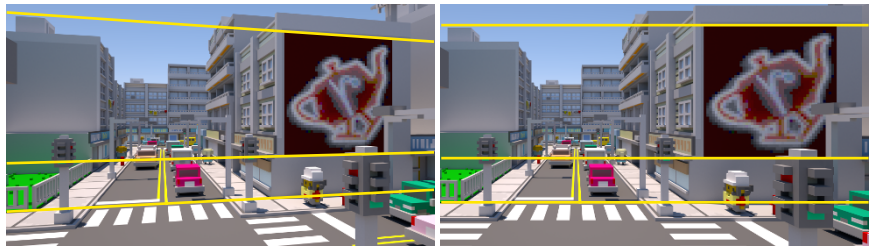
Figure 31-6. *Looking upward. (a) The original image, looking straight ahead. (b) Vertical lines are no longer parallel if the camera is turned to look upward. (c) Shifting as an off-center crop of a larger image; compare the solid and dashed green rectangles. (d) Shifting allows a higher view while keeping vertical lines parallel.*

(Figure 31-6b). This is a form of perspective distortion and is frequently called the *keystone effect*.

Instead, we can keep the view direction level as before and shift the sensor down relative to the lens. Alternatively, we can think of this as rendering to a larger image with an equivalently wider field of view (or zoomed out) and then cropping back to the original image size but off-center (Figure 31-6c).

This still allows us to see things higher up as with turning the camera, but now the lines on upright elements in our scene remain vertical (Figure 31-6d). With this, the keystoning or perspective distortion has been corrected.

Of course, we can also do the same thing horizontally. Here we have an example with the camera over the center of the street but turned to the right to avoid cutting off the billboard (Figure 31-7a). Instead, we can keep the camera pointed straight down the street and shift the sensor to the left so that we can capture more of the scene on the right side while keeping the horizontal lines parallel (Figure 31-7b).



(a) *Camera turned toward the right* **(b)** *Shifting toward the right*

Figure 31-7. *Looking to the side. (a) Turning toward the right. (b) Shifting toward the right.*

One other thing that this does is to simplify the two-point perspective of the first view down to a classic single-point perspective in the second. All of the converging lines now converge to a single point, which can be off-center. Careful use of shifting can also reduce a three-point perspective down to two-point while keeping the subject of a scene framed.

In view camera terms, a vertical shift is a "rise" or "fall," while a horizontal shift may be a "shift" or "cross."

31.4 LENS TILT

Next consider the case of a tilted lens. In view camera terms, this will be "swing" when horizontal and "tilt" when vertical. Of course, the two may be combined, and here the mathematics gets a little more complicated.

To model this, we will keep the lens centered at the origin in camera space, but the lens plane is now aligned with the unit normal vector $\hat{\mathbf{t}}$. See Figure 31-8.

Recall that the thin lens formula relates distances between the lens plane and the object and image points. These are perpendicular distances, so we need to use the scalar projections of the points onto vector $\hat{\mathbf{t}}$ (with appropriate sign corrections) to compute these distances.

Consequently, the mapping between a pixel position on the sensor, P', and the corresponding point on the focal plane, P, is still done through a scale factor (as in Equation 31.3), but now it is no longer constant:

$$P = P' \frac{f}{f + P' \cdot \hat{\mathbf{t}}}. \tag{31.5}$$

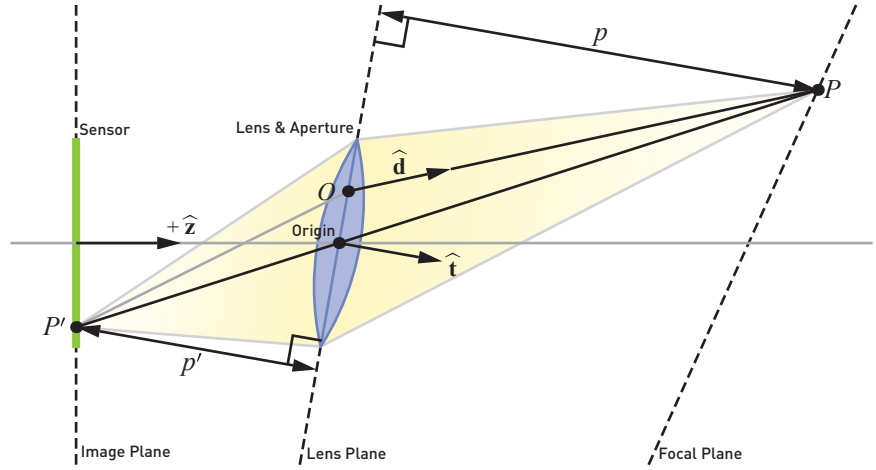


Figure 31-8. *The geometry of a tilted lens.*

Sampling the points on the lens to find the ray origin O is now slightly more complex and requires that we build an orthonormal basis in camera space around the tilt direction $\hat{\mathbf{t}}$.

Finding the ray direction remains mostly as before. However, because the distances between the planes are now varying when the lens is tilted, the focal plane and lens plane can now intersect. As a result, some points on the sensor may map to points that are behind the lens plane. When $\hat{\mathbf{t}} \cdot P < 0$, we have a virtual image. In this case, the rays diverge outward from the point behind the lens, and we must flip $\hat{\mathbf{d}}$ so that the ray goes forward into the scene.

The optics of the tilted lens has some interesting implications. The *Scheimpflug principle* states that when the image, lens, and focal planes are not parallel, then they all form a sheaf of planes that intersect along a common line, typically called the *Scheimpflug line* (Figure 31-9). Furthermore, the depth of field becomes a wedge, emanating from a second line, sometimes called the *hinge line*. As the sensor moves closer to or farther from the lens, the focal plane will pivot around this hinge line, forming a secondary sheaf of planes.

31.5 DIRECTING THE TILT

Because of all this, we have the flexibility to place the plane of focus nearly anywhere. But actually doing this through direct control of the lens tilt can be

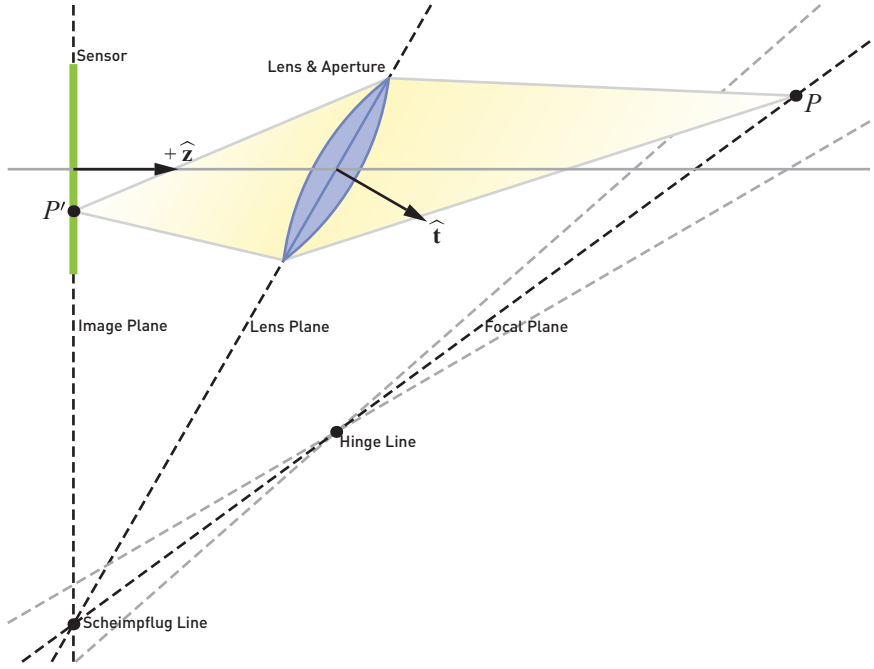


Figure 31-9. *The Scheimpflug principle and hinge line. The image, lens, and focal planes all intersect at a common line, which is shown here perpendicular to the page. The hinge line is also perpendicular to the page.*

tricky. Photographers tend to resort to tedious trial and error. In the digital world we can do better.

Suppose that we have three points, A, B, and C, defining the plane that we would like to have in focus (Figure 31-10). This plane has the unit normal vector $\hat{\mathbf{n}}$. How can we solve for the lens tilt $\hat{\mathbf{t}}$ and the distance s from the lens center to the image plane?

We know that points A and B must focus to points at the same z-coordinate behind the lens, which from the Equation 31.5 gives us s:

$$s = A_z \frac{f}{A \cdot \hat{\mathbf{t}} - f} = B_z \frac{f}{B \cdot \hat{\mathbf{t}} - f}. \tag{31.6}$$

Taking the right-hand sides, cross multiplying, expanding, and collecting gives

$$(A_z B_x - A_x B_z) t_x + (A_z B_y - A_y B_z) t_y = (A_z - B_z) f. \tag{31.7}$$

We also know from the Scheimpflug principle that the image plane, focal

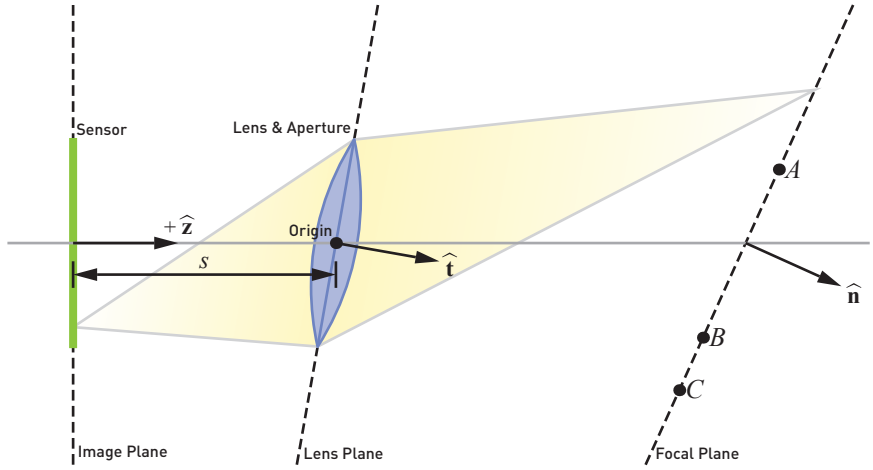


Figure 31-10. *Directing the lens tilt. Points A, B, and C are used to define the desired focal plane and appear collinear in a 2D projection such as this when the Scheimpflug line is drawn perpendicular to the page.*

plane, and lens plane must intersect at a line in common, which we can express as

$$\hat{\mathbf{z}} \cdot (\hat{\mathbf{n}} \times \hat{\mathbf{t}}) = 0. \tag{31.8}$$

Expanding this allows us to compute t_y in terms of t_x:

$$t_y = t_x n_y / n_x. \tag{31.9}$$

Taking Equation 31.9 and substituting it into Equation 31.7 allows us to solve for the x-component of the tilt, and from there we can compute the rest. Because our premises assumed that $\hat{\mathbf{t}}$ was scaled to unit length, the z-component must complete the unit vector. The full solution to the tilt of the lens and the placement of the sensor is thus:

$$t_x = \frac{(A_z - B_z)f}{A_z B_x - B_z A_x + (A_z B_y - B_z A_y) n_y / n_x}, \tag{31.10}$$

$$t_y = t_x n_y / n_x, \tag{31.11}$$

$$t_z = \sqrt{1 - t_x^2 - t_y^2}, \tag{31.12}$$

$$s = A_z \frac{f}{A \cdot \hat{\mathbf{t}} - f}. \tag{31.13}$$

In some cases, depending on the desired plane of focus, $|n_x|$ may become quite small or even be zero. To avoid division by zero or other numeric

precision problems, the x- and y-components can be exchanged by symmetry if n_x is closer to zero than n_y. However, if both n_x and n_y are zero, then the plane of focus is parallel to the image and lens planes and there is no lens tilt. In this case, setting t_x and t_y to zero before computing t_z and s produces the correct results, and the tilted lens model reduces to the standard thin lens model.

Combining both the shift and tilt extensions to the thin lens camera model, together with the solutions to find the tilt and the shift positions, yields the following implementation of the full tilt-shift model:

```
void tilt_shift(vec2 screen, vec2 random,
                out vec3 ray_origin, out vec3 ray_direction)
{
    // n  : normal      A : focus_a
    // t  : tilt        B : focus_b
    // M  : middle      C : focus_c
    // M' : shift

    // Focal plane values (precomputable)
    vec3 normal = normalize(cross(focus_b - focus_a,
                                  focus_c - focus_a));
    // Lens values (precomputable)
    vec3 tilt = vec3(0.0);
    if (abs(normal.x) > abs(normal.y))
    {
        tilt.x = (focus_a.z - focus_b.z) * focal_length /
            (focus_a.z * focus_b.x - focus_b.z * focus_a.x +
             (focus_a.z * focus_b.y - focus_b.z * focus_a.y) *
             normal.y / normal.x);
        tilt.y = tilt.x * normal.y / normal.x;
    }
    else if (abs(normal.y) > 0.0)
    {
        tilt.y = (focus_a.z - focus_b.z) * focal_length /
            (focus_a.z * focus_b.y - focus_b.z * focus_a.y +
             (focus_a.z * focus_b.x - focus_b.z * focus_a.x) *
             normal.x / normal.y);
        tilt.x = tilt.y * normal.x / normal.y;
    }
    tilt.z = sqrt(1.0 - tilt.x * tilt.x - tilt.y * tilt.y);
    vec3 basis_u = normalize(cross(tilt,
        abs(tilt.x) > abs(tilt.y) ? vec3(0.0, 1.0, 0.0)
                                  : vec3(1.0, 0.0, 0.0)));
    vec3 basis_v = cross(tilt, basis_u);
    float aperture = focal_length / f_stop;
    // Image plane values (precomputable)
    float image_plane = focus_a.z * focal_length /
        (dot(focus_a, tilt) - focal_length);
    vec2 shift = middle.xy / middle.z * -image_plane;

    // Image plane values (render-time)
    vec3 sensor = vec3(screen * 0.5 * sensor_size + shift,
                       -image_plane);
```

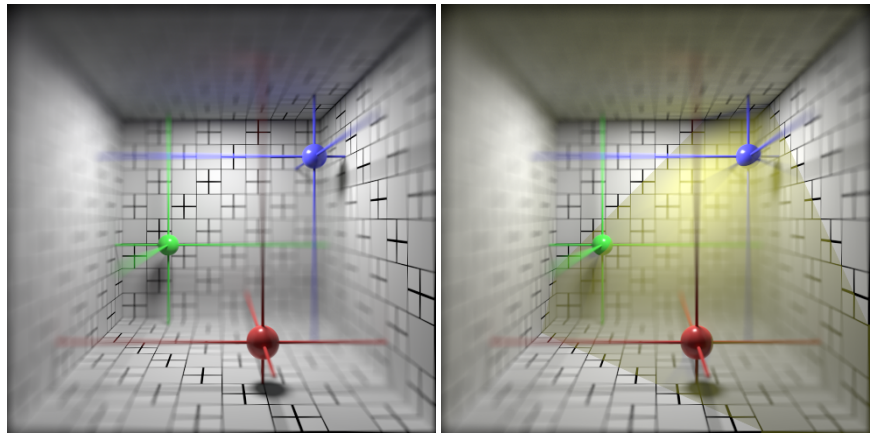
```
    // Lens values (render-time)
    float theta = 6.28318531 * random.x;
    float r = 0.5 * aperture * sqrt(random.y);
    vec3 lens = (cos(theta) * basis_u +
                 sin(theta) * basis_v) * r;
    // Focal plane values (render-time)
    vec3 focused = sensor * focal_length /
        (focal_length + dot(sensor, tilt));
    float flip = sign(dot(tilt, focused));

    ray_origin = lens;
    ray_direction = flip * normalize(focused - lens);
}
```

31.6 RESULTS

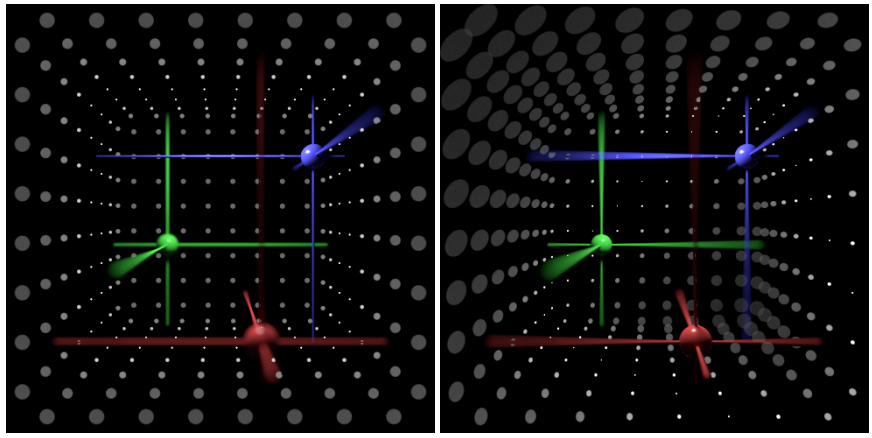
Figure 31-11 shows a test scene with the tilted lens in action and where the lens orientation and sensor distance has been solved for using the centers of the three spheres as our three points. All three points, despite being at different distances from the camera, are in perfect focus. Moreover, the focal plane intersects the box and produces a sharp appearance along diagonals in the grid texture. Although this is a relatively simple scene, already the defocus is much more visually complex than a simple blur masked along a gradient.

With the computer solving the tilt for us on each frame, we can easily animate it. For example, we can keep any three subjects in focus as they move about while the rest of the scene is de-emphasized and out of focus. Alternatively, we could hold the focal plane fixed in world space while moving the camera around. There are many possibilities for rendering with tilt-shift effects.



(a) *Three points defining the focus* **(b)** *Focal plane highlighted in yellow*

Figure 31-11. *Focus aligned to the plane through the spheres. (a) Without the plane of focus shown. (b) With the plane of focus highlighted in yellow.*



(a) *Standard lens* **(b)** *Tilted lens*

Figure 31-12. *Bokeh. (a) Circular bokeh with a standard thin lens focused on the blue sphere. (b) Elliptical bokeh with a tilted lens focused on all three spheres.*

Another interesting effect predicted by our tilt-shift model pertains to bokeh. Figure 31-12a shows a scene similar to the previous one except with the grid texture of the box replaced with very small, bright points and rendered with the non-tilted lens of the standard thin lens camera model focused on the blue sphere at middle distance. In this case, the small bright points of the surrounding box are circular bokeh, as usual.

However, if we return to the lens tilt from before with the plane of focus passing through the center of the three spheres, then some of the bokeh elongate into ellipses (Figure 31-12b). This should make some intuitive sense: from the perspective of the light cones from each bright point passing through the lens, the image sensor now cuts through at an oblique angle. In other words, the bokeh form conic sections.

The tilt-shift model presented in this chapter is incorporated in the production camera projection plugin included with Pixar's RenderMan. Though all of the rendered figures shown in this chapter were rendered with RenderMan, a complete self-contained demonstration that incorporates the sample code listings for the `thin_lens()` and `tilt_shift()` functions and recreates the scenes from Figure 31-11a and Figure 31-12 can be found online on Shadertoy at https://www.shadertoy.com/view/tlcBzN and at the book's source code website.

ACKNOWLEDGMENTS

Thanks to the RenderMan team at Pixar Animation Studios for supporting this work. It was done prior to the author joining Amazon. The city scene uses artwork assets from the "Mini Mike's Metro Minis" collection by Mike Judge, used under the Creative Commons Attribution 4.0 International license.

REFERENCES

[1] Cook, R. L., Porter, T., and Carpenter, L. Distributed ray tracing. *SIGGRAPH Computer Graphics*, 18(3):137–145, 1984.

[2] Merklinger, H. M. *Focusing the View Camera*. v. 1.6.1 first internet edition, 2010. http://www.trenholm.org/hmmerk/FVC161.pdf.

[3] Potmesil, M. and Chakravarty, I. A lens and aperture camera model for synthetic image generation. *SIGGRAPH Computer Graphics*, 15(3):297–305, 1981.