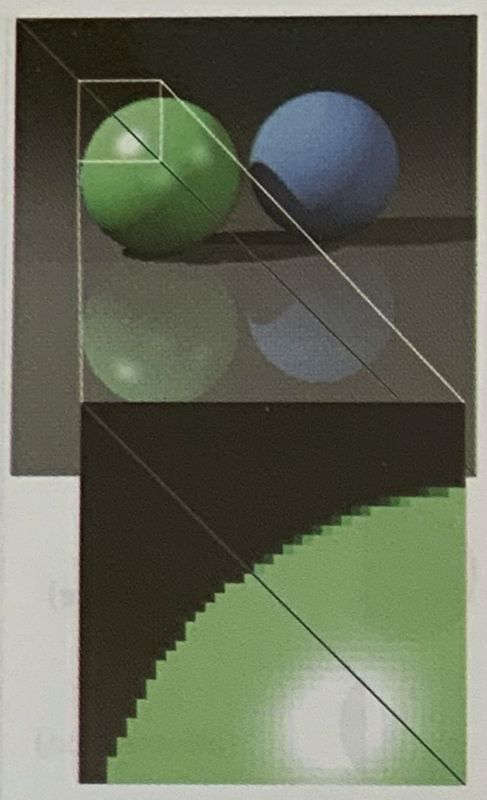

Figure 13.8. A simple scene rendered with one sample per pixel (lower left half) and nine samples per pixel (upper right half).

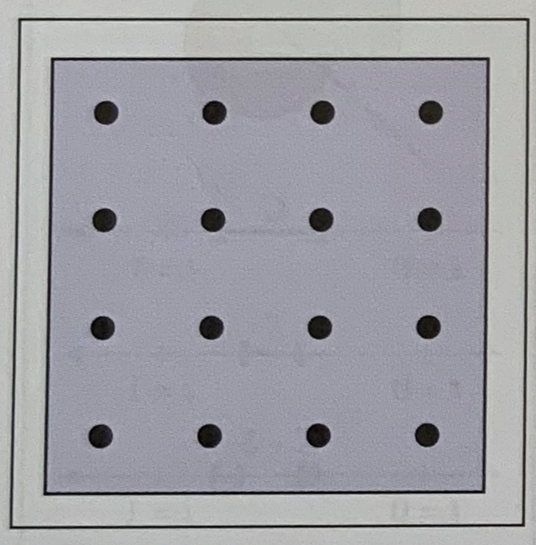

Figure 13.9. Sixteen regular samples for a single pixel.

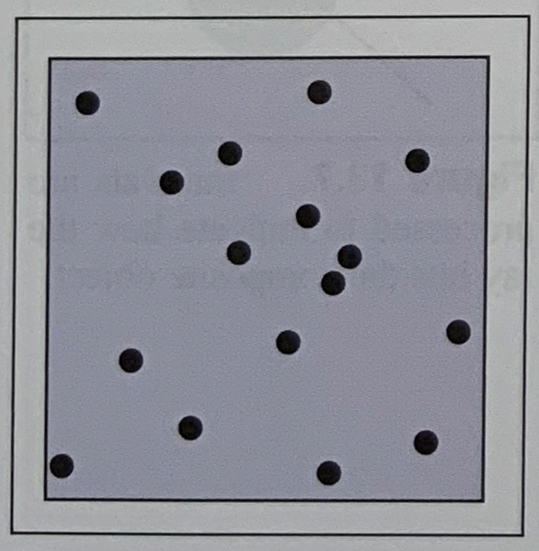

Figure 13.10. Sixteen random samples for a single pixel.

13.4.1 Antialiasing

Recall that a simple way to antialias an image is to compute the average color for the area of the pixel rather than the color at the center point. In ray tracing, our computational primitive is to compute the color at a point on the screen. If we average many of these points across the pixel, we are approximating the true average. If the screen coordinates bounding the pixel are $[i, i+1] \times [j, j+1]$, then we can replace the loop:

for each pixel (i, j) **do**
$c_{ij} =$ ray-color$(i + 0.5, j + 0.5)$

with code that samples on a regular $n \times n$ grid of samples within each pixel:

for each pixel (i, j) **do**
$c = 0$
for $p = 0$ to $n - 1$ **do**
for $q = 0$ to $n - 1$ **do**
$c = c +$ ray-color$(i + (p + 0.5)/n, j + (q + 0.5)/n)$
$c_{ij} = c/n^2$

This is usually called *regular sampling*. The 16 sample locations in a pixel for $n = 4$ are shown in Figure 13.9. Note that this produces the same answer as rendering a traditional ray-traced image with one sample per pixel at $n_x n$ by $n_y n$ resolution and then averaging blocks of n by n pixels to get a n_x by n_y image.

One potential problem with taking samples in a regular pattern within a pixel is that regular artifacts such as moiré patterns can arise. These artifacts can be turned into noise by taking samples in a random pattern within each pixel as shown in Figure 13.10. This is usually called *random sampling* and involves just a small change to the code:

for each pixel (i, j) **do**
$c = 0$
for $p = 1$ to n^2 **do**
$c = c +$ ray-color$(i + \xi, j + \xi)$
$c_{ij} = c/n^2$

Here ξ is a call that returns a uniform random number in the range $[0, 1)$. Unfortunately, the noise can be quite objectionable unless many samples are taken. A compromise is to make a hybrid strategy that randomly perturbs a regular grid:

for each pixel (i, j) **do**
$c = 0$
for $p = 0$ to $n - 1$ **do**

for $q = 0$ to $n - 1$ **do**
$c = c +$ ray-color$(i + (p + \xi)/n, j + (q + \xi)/n)$
$c_{ij} = c/n^2$

That method is usually called *jittering* or *stratified sampling* (Figure 13.11).

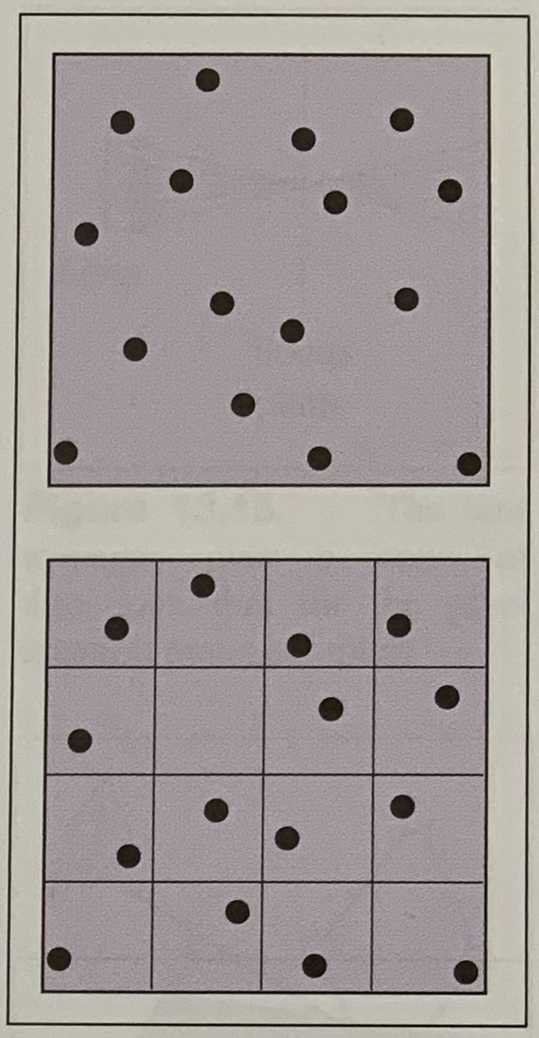

Figure 13.11. Sixteen stratified (jittered) samples for a single pixel shown with and without the bins highlighted. There is exactly one random sample taken within each bin.

13.4.2 Soft Shadows

The reason shadows are hard to handle in standard ray tracing is that lights are infinitesimal points or directions and are thus either visible or invisible. In real life, lights have nonzero area and can thus be partially visible. This idea is shown in 2D in Figure 13.12. The region where the light is entirely invisible is called the *umbra*. The partially visible region is called the *penumbra*. There is not a commonly used term for the region not in shadow, but it is sometimes called the *anti-umbra*.

The key to implementing soft shadows is to somehow account for the light being an area rather than a point. An easy way to do this is to approximate the light with a distributed set of N point lights each with one Nth of the intensity of the base light. This concept is illustrated at the left of Figure 13.13 where nine lights are used. You can do this in a standard ray tracer, and it is a common trick to get soft shadows in an off-the-shelf renderer. There are two potential problems with this technique. First, typically dozens of point lights are needed to achieve visually smooth results, which slows down the program a great deal. The second problem is that the shadows have sharp transitions inside the penumbra.

Distribution ray tracing introduces a small change in the shadowing code. Instead of representing the area light at a discrete number of point sources, we represent it as an infinite number and choose one at random for each viewing ray. This amounts to choosing a random point on the light for any surface point being lit as is shown at the right of Figure 13.13.

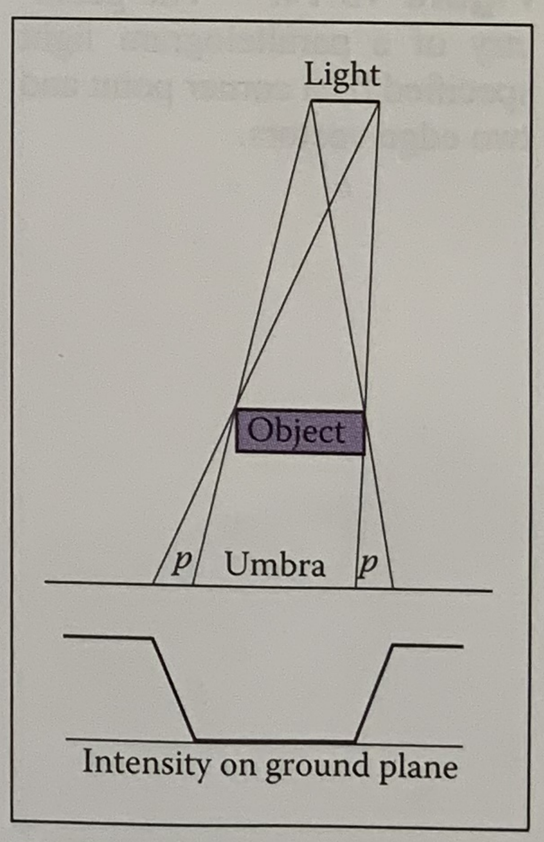

Figure 13.12. A soft shadow has a gradual transition from the unshadowed to shadowed region. The transition zone is the "penumbra" denoted by p in the figure.

If the light is a parallelogram specified by a corner point $\mathbf{c}$ and two edge vectors $\mathbf{a}$ and $\mathbf{b}$ (Figure 13.14), then choosing a random point $\mathbf{r}$ is straightforward:

$$\mathbf{r} = \mathbf{c} + \xi_1 \mathbf{a} + \xi_2 \mathbf{b},$$

where ξ_1 and ξ_2 are uniform random numbers in the range $[0, 1)$.

We then send a shadow ray to this point as shown at the right in Figure 13.13. Note that the direction of this ray is not unit length, which may require some modification to your basic ray tracer depending upon its assumptions.

We would really like to jitter points on the light. However, it can be dangerous to implement this without some thought. We would not want to always have the

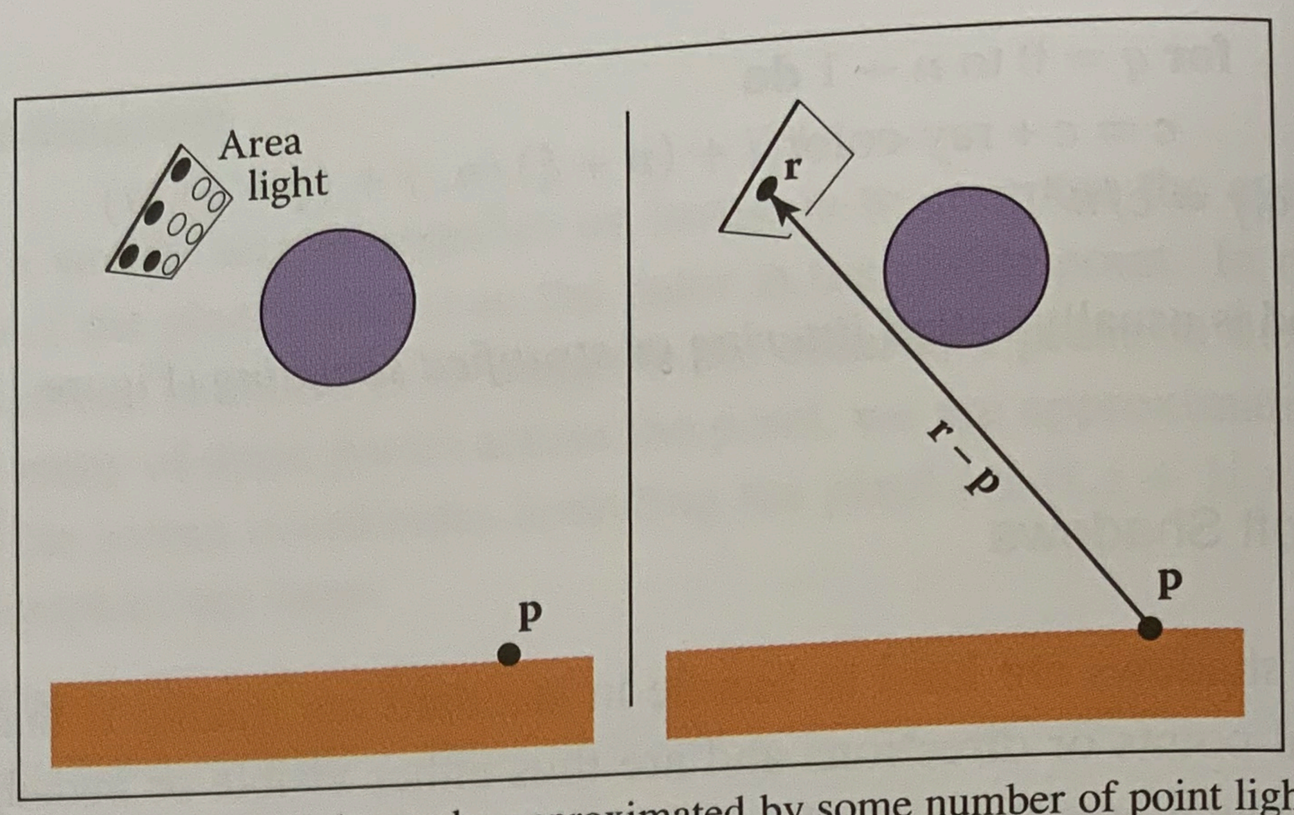

Figure 13.13. Left: an area light can be approximated by some number of point lights; four of the nine points are visible to **p** so it is in the penumbra. Right: a random point on the light is chosen for the shadow ray, and it has some chance of hitting the light or not.

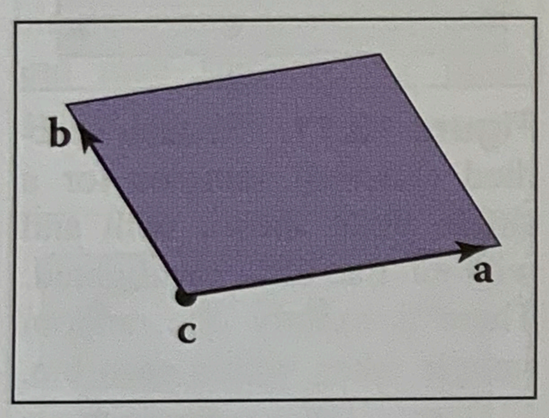

Figure 13.14. The geometry of a parallelogram light specified by a corner point and two edge vectors.

ray in the upper left-hand corner of the pixel generate a shadow ray to the upper left-hand corner of the light. Instead we would like to scramble the samples, such that the pixel samples and the light samples are each themselves jittered, but so that there is no correlation between pixel samples and light samples. A good way to accomplish this is to generate two distinct sets of n^2 jittered samples and pass samples into the light source routine:

for each pixel (i, j) **do**
  $c = 0$
  generate $N = n^2$ jittered 2D points and store in array r[ ]
  generate $N = n^2$ jittered 2D points and store in array s[ ]
  shuffle the points in array s[ ]
  for $p = 0$ to $N - 1$ **do**
    $c = c +$ ray-color$(i +$ r[p].x(), $j +$ r[p].y(), s[p])
  $c_{ij} = c/N$

This shuffle routine eliminates any coherence between arrays r and s. The shadow routine will just use the 2D random point stored in $s[p]$ rather than calling the random number generator. A shuffle routine for an array indexed from 0 to $N-1$ is:

for $i = N - 1$ downto 1 **do**
  choose random integer j between 0 and i inclusive
  swap array elements i and j

13.4.3 Depth of Field

The soft focus effects seen in most photos can be simulated by collecting light at a nonzero size "lens" rather than at a point. This is called *depth of field*. The

lens collects light from a cone of directions that has its apex at a distance where everything is in focus (Figure 13.15). We can place the "window" we are sampling on the plane where everything is in focus (rather than at the $z = n$ plane as we did previously) and the lens at the eye. The distance to the plane where everything is in focus we call the *focus plane*, and the distance to it is set by the user, just as the distance to the focus plane in a real camera is set by the user or range finder.

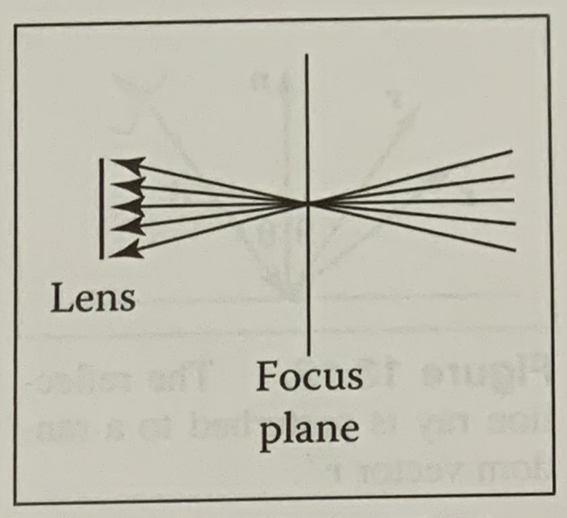

Figure 13.15. The lens averages over a cone of directions that hit the pixel location being sampled.

To be most faithful to a real camera, we should make the lens a disk. However, we will get very similar effects with a square lens (Figure 13.16). So we choose the side-length of the lens and take random samples on it. The origin of the view rays will be these perturbed positions rather than the eye position. Again, a shuffling routine is used to prevent correlation with the pixel sample positions. An example using 25 samples per pixel and a large disk lens is shown in Figure 13.17.

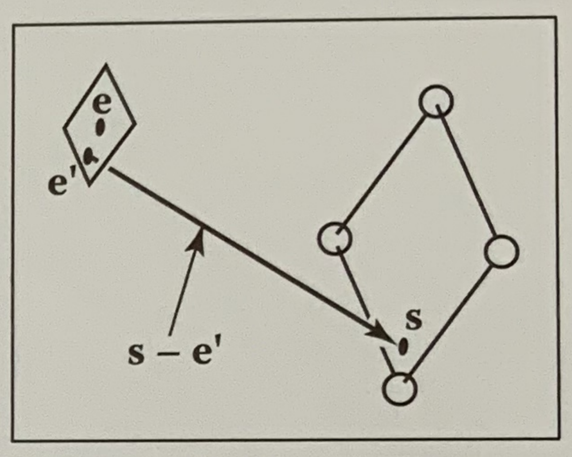

Figure 13.16. To create depth-of-field effects, the eye is randomly selected from a square region.

13.4.4 Glossy Reflection

Some surfaces, such as brushed metal, are somewhere between an ideal mirror and a diffuse surface. Some discernible image is visible in the reflection, but it is blurred. We can simulate this by randomly perturbing ideal specular reflection rays as shown in Figure 13.18.

Only two details need to be worked out: how to choose the vector $\mathbf{r}'$ and what to do when the resulting perturbed ray is below the surface from which the ray is

Figure 13.17. An example of depth of field. The caustic in the shadow of the wine glass is computed using particle tracing as described in Chapter 23.

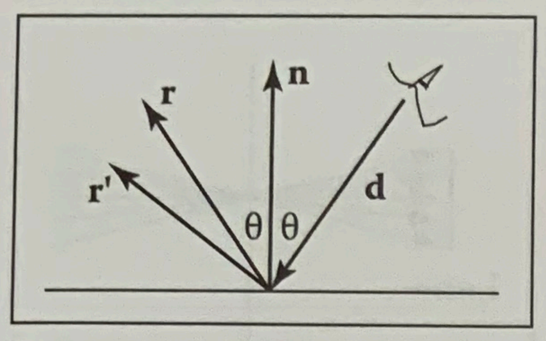

Figure 13.18. The reflection ray is perturbed to a random vector $\mathbf{r}'$.

reflected. The latter detail is usually settled by returning a zero color when the ray is below the surface.

To choose $\mathbf{r}'$, we again sample a random square. This square is perpendicular to $\mathbf{r}$ and has width a which controls the degree of blur. We can set up the square's orientation by creating an orthonormal basis with $\mathbf{w} = \mathbf{r}$ using the techniques in Section 2.4.6. Then, we create a random point in the 2D square with side length a centered at the origin. If we have 2D sample points $(\xi, \xi') \in [0, 1]^2$, then the analogous point on the desired square is

$$u = -\frac{a}{2} + \xi a,$$

$$v = -\frac{a}{2} + \xi' a.$$

Because the square over which we will perturb is parallel to both the $\mathbf{u}$ and $\mathbf{v}$ vectors, the ray $\mathbf{r}'$ is just

$$\mathbf{r}' = \mathbf{r} + u\mathbf{u} + v\mathbf{v}.$$

Note that $\mathbf{r}'$ is not necessarily a unit vector and should be normalized if your code requires that for ray directions.

13.4.5 Motion Blur

We can add a blurred appearance to objects as shown in Figure 13.19. This is called *motion blur* and is the result of the image being formed over a nonzero

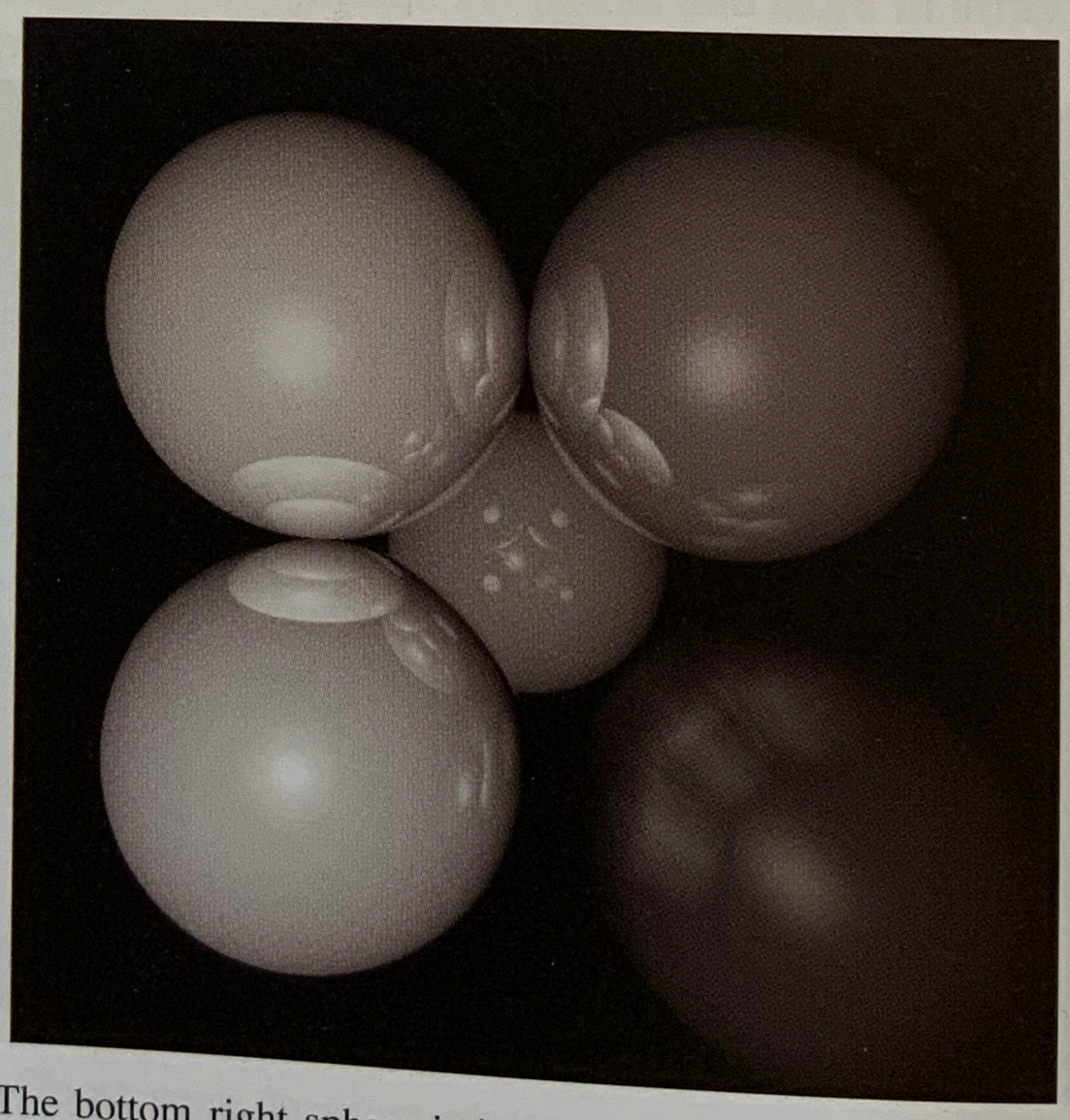

Figure 13.19. The bottom right sphere is in motion, and a blurred appearance results. *Image courtesy Chad Barb.*

span of time. In a real camera, the aperture is open for some time interval during which objects move. We can simulate the open aperture by setting a time variable ranging from T_0 to T_1. For each viewing ray we choose a random time,

$$T = T_0 + \xi(T_1 - T_0).$$

We may also need to create some objects to move with time. For example, we might have a moving sphere whose center travels from $\mathbf{c}_0$ to $\mathbf{c}_1$ during the interval. Given T, we could compute the actual center and do a ray–intersection with that sphere. Because each ray is sent at a different time, each will encounter the sphere at a different position, and the final appearance will be blurred. Note that the bounding box for the moving sphere should bound its entire path so an efficiency structure can be built for the whole time interval (Glassner, 1988).

Notes

There are many, many other advanced methods that can be implemented in the ray-tracing framework. Some resources for further information are Glassner's *An Introduction to Ray Tracing* and *Principles of Digital Image Synthesis*, Shirley's *Realistic Ray Tracing*, and Pharr and Humphreys's *Physically Based Rendering: From Theory to Implementation*.

Frequently Asked Questions

- What is the best ray-intersection efficiency structure?

The most popular structures are binary space partitioning trees (BSP trees), uniform subdivision grids, and bounding volume hierarchies. Most people who use BSP trees make the splitting planes axis-aligned, and such trees are usually called k-d trees. There is no clear-cut answer for which is best, but all are much, much better than brute-force search in practice. If I were to implement only one, it would be the bounding volume hierarchy because of its simplicity and robustness.

- Why do people use bounding boxes rather than spheres or ellipsoids?

Sometimes spheres or ellipsoids are better. However, many models have polygonal elements that are tightly bounded by boxes, but they would be difficult to tightly bind with an ellipsoid.