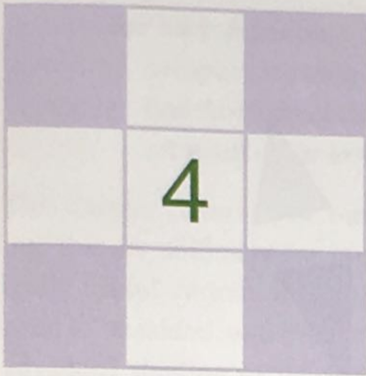


4 Ray Tracing

One of the basic tasks of computer graphics is *rendering* three-dimensional objects: taking a scene, or model, composed of many geometric objects arranged in 3D space and producing a 2D image that shows the objects as viewed from a particular viewpoint. It is the same operation that has been done for centuries by architects and engineers creating drawings to communicate their designs to others.

Fundamentally, rendering is a process that takes as its input a set of objects and produces as its output an array of pixels. One way or another, rendering involves considering how each object contributes to each pixel; it can be organized in two general ways. In *object-order rendering*, each object is considered in turn, and for each object all the pixels that it influences are found and updated. In *image-order rendering*, each pixel is considered in turn, and for each pixel all the objects that influence it are found and the pixel value is computed. You can think of the difference in terms of the nesting of loops: in image-order rendering the "for each pixel" loop is on the outside, whereas in object-order rendering the "for each object" loop is on the outside.

If the output is a vector image rather than a raster image, rendering doesn't have to involve pixels, but we'll assume raster images in this book.

Image-order and object-order rendering approaches can compute exactly the same images, but they lend themselves to computing different kinds of effects and have quite different performance characteristics. We'll explore the comparative strengths of the approaches in Chapter 8 after we have discussed them both, but, broadly speaking, image-order rendering is simpler to get working and more flexible in the effects that can be produced, and usually (though not always) takes much more execution time to produce a comparable image.

In a ray tracer, it is easy to compute accurate shadows and reflections, which are awkward in the object-order framework.

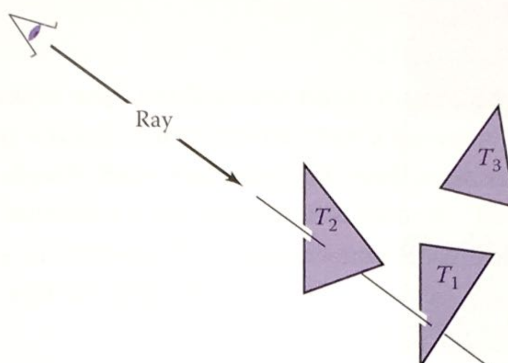


Figure 4.1. The ray is "traced" into the scene and the first object hit is the one seen through the pixel. In this case, the triangle T_2 is returned.

Ray tracing is an image-order algorithm for making renderings of 3D scenes, and we'll consider it first because it's possible to get a ray tracer working without developing any of the mathematical machinery that's used for object-order rendering.

4.1 The Basic Ray-Tracing Algorithm

A ray tracer works by computing one pixel at a time, and for each pixel the basic task is to find the object that is seen at that pixel's position in the image. Each pixel "looks" in a different direction, and any object that is seen by a pixel must intersect the *viewing ray*, a line that emanates from the viewpoint in the direction that pixel is looking. The particular object we want is the one that intersects the viewing ray nearest the camera, since it blocks the view of any other objects behind it. Once that object is found, a *shading* computation uses the intersection point, surface normal, and other information (depending on the desired type of rendering) to determine the color of the pixel. This is shown in Figure 4.1, where the ray intersects two triangles, but only the first triangle hit, T_2, is shaded.

A basic ray tracer therefore has three parts:

1. *ray generation*, which computes the origin and direction of each pixel's viewing ray based on the camera geometry;
2. *ray intersection*, which finds the closest object intersecting the viewing ray;
3. *shading*, which computes the pixel color based on the results of ray intersection.

The structure of the basic ray tracing program is:

```
for each pixel do
    compute viewing ray
    find first object hit by ray and its surface normal n
    set pixel color to value computed from hit point, light, and n
```

This chapter covers basic methods for ray generation, ray intersection, and shading, that are sufficient for implementing a simple demonstration ray tracer. For a really useful system, more efficient ray intersection techniques from Chapter 12 need to be added, and the real potential of a ray tracer will be seen with the more advanced shading methods from Chapter 10 and the additional rendering techniques from Chapter 13.

4.2 Perspective

The problem of representing a 3D object or scene with a 2D drawing or painting was studied by artists hundreds of years before computers. Photographs also represent 3D scenes with 2D images. While there are many unconventional ways to make images, from cubist painting to fisheye lenses (Figure 4.2) to peripheral cameras, the standard approach for both art and photography, as well as computer graphics, is *linear perspective*, in which 3D objects are projected onto an *image plane* in such a way that straight lines in the scene become straight lines in the image.

Figure 4.2. An image taken with a fisheye lens is not a linear perspective image. *Photo courtesy Philip Greenspun.*

The simplest type of projection is *parallel projection*, in which 3D points are mapped to 2D by moving them along a *projection direction* until they hit the image plane (Figures 4.3–4.4). The view that is produced is determined by the choice of projection direction and image plane. If the image plane is perpendicular

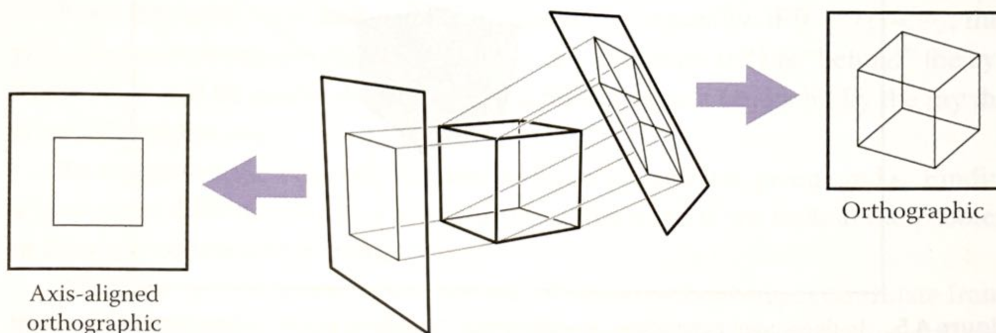


Figure 4.3. When projection lines are parallel and perpendicular to the image plane, the resulting views are called orthographic.

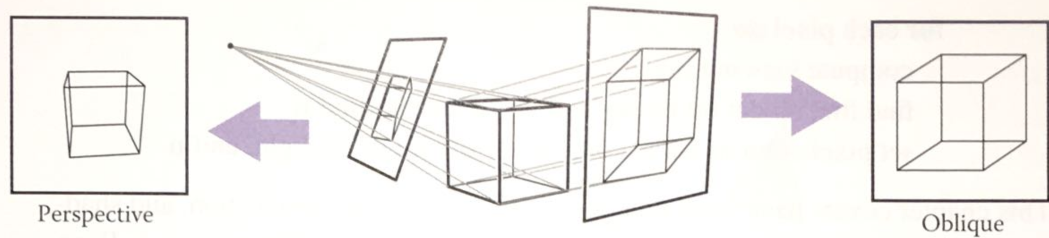


Figure 4.4. A parallel projection that has the image plane at an angle to the projection direction is called oblique (right). In perspective projection, the projection lines all pass through the viewpoint, rather than being parallel (left). The illustrated perspective view is non-oblique because a projection line drawn through the center of the image would be perpendicular to the image plane.

to the view direction, the projection is called *orthographic*; otherwise it is called *oblique*.

Some books reserve "orthographic" for projection directions that are parallel to the coordinate axes.

Parallel projections are often used for mechanical and architectural drawings because they keep parallel lines parallel and they preserve the size and shape of planar objects that are parallel to the image plane.

The advantages of parallel projection are also its limitations. In our everyday experience (and even more so in photographs) objects look smaller as they get farther away, and as a result parallel lines receding into the distance do not appear parallel. This is because eyes and cameras don't collect light from a single viewing direction; they collect light that passes through a particular viewpoint. As has been recognized by artists since the Renaissance, we can produce natural-looking views using *perspective projection*: we simply project along lines that pass through a single point, the *viewpoint*, rather than along parallel lines (Fig-

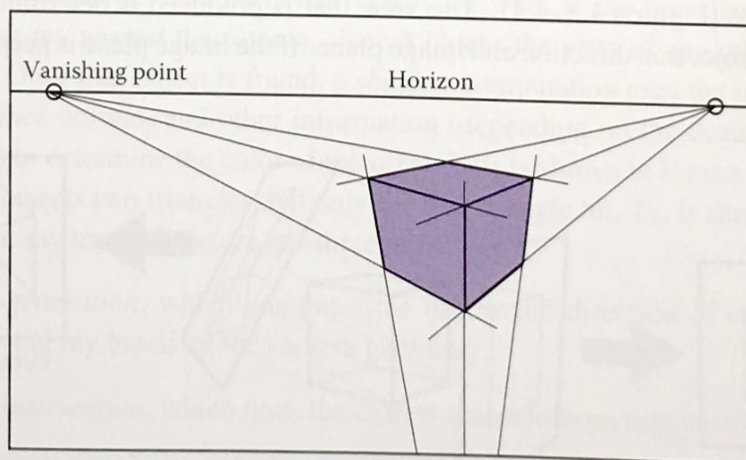


Figure 4.5. In three-point perspective, an artist picks "vanishing points" where parallel lines meet. Parallel horizontal lines will meet at a point on the horizon. Every set of parallel lines has its own vanishing points. These rules are followed automatically if we implement perspective based on the correct geometric principles.

ure 4.4). In this way, objects farther from the viewpoint naturally become smaller when they are projected. A perspective view is determined by the choice of viewpoint (rather than projection direction) and image plane. As with parallel views, there are oblique and non-oblique perspective views; the distinction is made based on the projection direction at the center of the image.

You may have learned about the artistic conventions of *three-point perspective*, a system for manually constructing perspective views (Figure 4.5). A surprising fact about perspective is that all the rules of perspective drawing will be followed automatically if we follow the simple mathematical rule underlying perspective: objects are projected directly toward the eye, and they are drawn where they meet a view plane in front of the eye.

4.3 Computing Viewing Rays

From the previous section, the basic tools of ray generation are the viewpoint (or view direction, for parallel views) and the image plane. There are many ways to work out the details of camera geometry; in this section we explain one based on orthonormal bases that supports normal and oblique parallel and orthographic views.

In order to generate rays, we first need a mathematical representation for a ray. A ray is really just an origin point and a propagation direction; a 3D parametric line is ideal for this. As discussed in Section 2.5.7, the 3D parametric line from the eye $\mathbf{e}$ to a point $\mathbf{s}$ on the image plane (Figure 4.6) is given by

$$\mathbf{p}(t) = \mathbf{e} + t(\mathbf{s} - \mathbf{e}).$$

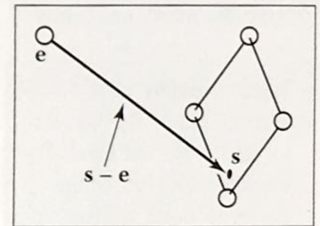


Figure 4.6. The ray from the eye to a point on the image plane.

This should be interpreted as, "we advance from $\mathbf{e}$ along the vector $(\mathbf{s} - \mathbf{e})$ a fractional distance t to find the point $\mathbf{p}$." So given t, we can determine a point $\mathbf{p}$. The point $\mathbf{e}$ is the ray's *origin*, and $\mathbf{s} - \mathbf{e}$ is the ray's *direction*.

Note that $\mathbf{p}(0) = \mathbf{e}$, and $\mathbf{p}(1) = \mathbf{s}$, and more generally, if $0 < t_1 < t_2$, then $\mathbf{p}(t_1)$ is closer to the eye than $\mathbf{p}(t_2)$. Also, if $t < 0$, then $\mathbf{p}(t)$ is "behind" the eye. These facts will be useful when we search for the closest object hit by the ray that is not behind the eye.

Caution: we are overloading the variable t, which is the ray parameter and also the v-coordinate of the top edge of the image.

To compute a viewing ray, we need to know $\mathbf{e}$ (which is given) and $\mathbf{s}$. Finding $\mathbf{s}$ may seem difficult, but it is actually straightforward if we look at the problem in the right coordinate system.

All of our ray-generation methods start from an orthonormal coordinate frame known as the *camera frame*, which we'll denote by $\mathbf{e}$, for the eye point, or viewpoint, and $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ for the three basis vectors, organized with $\mathbf{u}$ pointing rightward (from the camera's view), $\mathbf{v}$ pointing upward, and $\mathbf{w}$ pointing backward, so

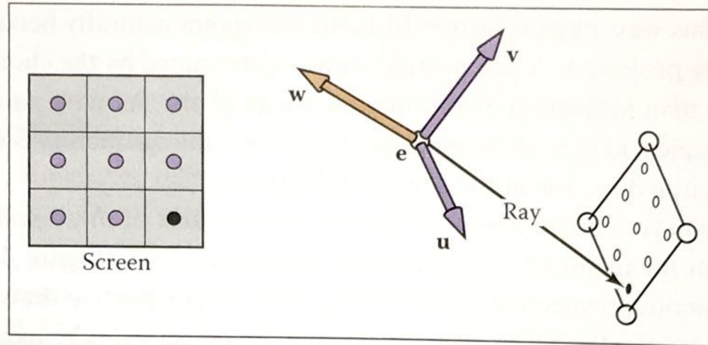


Figure 4.7. The sample points on the screen are mapped to a similar array on the 3D window. A viewing ray is sent to each of these locations.

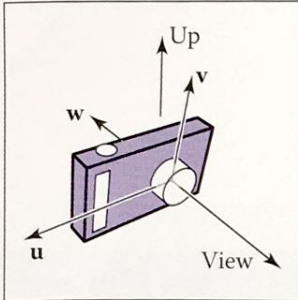


Figure 4.8. The vectors of the camera frame, together with the view direction and up direction. The **w** vector is opposite the view direction, and the **v** vector is coplanar with **w** and the up vector.

Since **v** and **w** have to be perpendicular, the up vector and **v** are not generally the same. But setting the up vector to point straight upward in the scene will orient the camera in the way we would think of as "upright."

It might seem logical that orthographic viewing rays should start from infinitely far away, but then it would not be possible to make orthographic views of an object inside a room, for instance.

Many systems assume that $l = -r$ and $b = -t$ so that a width and a height suffice.

that $\{\mathbf{u}, \mathbf{v}, \mathbf{w}\}$ forms a right-handed coordinate system. The most common way to construct the camera frame is from the viewpoint, which becomes $\mathbf{e}$, the *view direction*, which is $-\mathbf{w}$, and the *up vector*, which is used to construct a basis that has $\mathbf{v}$ and $\mathbf{w}$ in the plane defined by the view direction and the up direction, using the process for constructing an orthonormal basis from two vectors described in Section 2.4.7.

4.3.1 Orthographic Views

For an orthographic view, all the rays will have the direction $-\mathbf{w}$. Even though a parallel view doesn't have a viewpoint per se, we can still use the origin of the camera frame to define the plane where the rays start, so that it's possible for objects to be behind the camera.

The viewing rays should start on the plane defined by the point $\mathbf{e}$ and the vectors $\mathbf{u}$ and $\mathbf{v}$; the only remaining information required is *where* on the plane the image is supposed to be. We'll define the image dimensions with four numbers, for the four sides of the image: l and r are the positions of the left and right edges of the image, as measured from $\mathbf{e}$ along the $\mathbf{u}$ direction; and b and t are the positions of the bottom and top edges of the image, as measured from $\mathbf{e}$ along the $\mathbf{v}$ direction. Usually $l < 0 < r$ and $b < 0 < t$. (See Figure 4.9.)

In Section 3.2 we discussed pixel coordinates in an image. To fit an image with $n_x \times n_y$ pixels into a rectangle of size $(r - l) \times (t - b)$, the pixels are spaced a distance $(r - l)/n_x$ apart horizontally and $(t - b)/n_y$ apart vertically, with a half-pixel space around the edge to center the pixel grid within the image rectangle. This means that the pixel at position (i, j) in the raster image has the position

$$\begin{aligned} u &= l + (r - l)(i + 0.5)/n_x, \\ v &= b + (t - b)(j + 0.5)/n_y, \end{aligned} \tag{4.1}$$

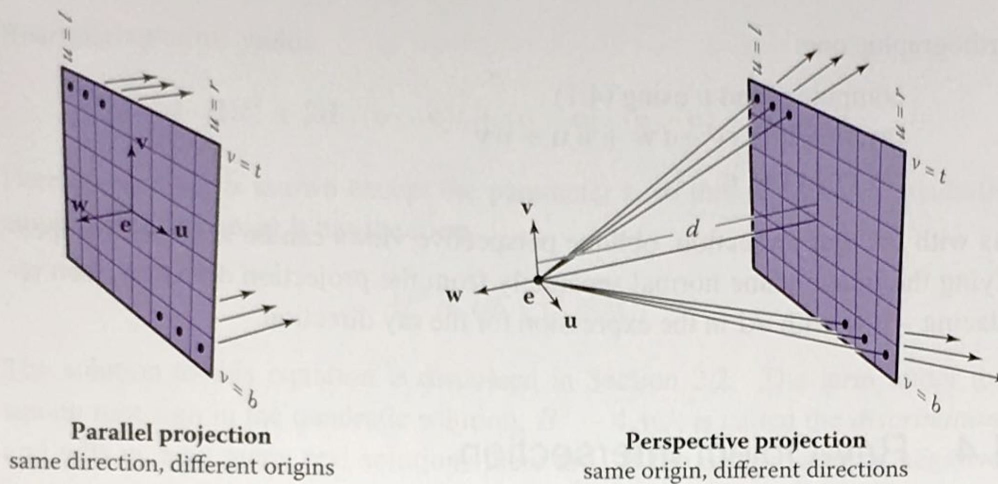


Figure 4.9. Ray generation using the camera frame. Left: In an orthographic view, the rays start at the pixels' locations on the image plane, and all share the same direction, which is equal to the view direction. Right: In a perspective view, the rays start at the viewpoint, and each ray's direction is defined by the line through the viewpoint, **e**, and the pixel's location on the image plane.

where (u, v) are the coordinates of the pixel's position on the image plane, measured with respect to the origin $\mathbf{e}$ and the basis $\{\mathbf{u}, \mathbf{v}\}$.

In an orthographic view, we can simply use the pixel's image-plane position as the ray's starting point, and we already know the ray's direction is the view direction. The procedure for generating orthographic viewing rays is then:

compute u and v using (4.1)
ray.direction $\leftarrow -\mathbf{w}$
ray.origin $\leftarrow \mathbf{e} + u\,\mathbf{u} + v\,\mathbf{v}$

With l and r both specified, there is redundancy: moving the viewpoint a bit to the right and correspondingly decreasing l and r will not change the view (and similarly on the $\mathbf{v}$-axis).

It's very simple to make an oblique parallel view: just allow the image plane normal $\mathbf{w}$ to be specified separately from the view direction $\mathbf{d}$. The procedure is then exactly the same, but with $\mathbf{d}$ substituted for $-\mathbf{w}$. Of course $\mathbf{w}$ is still used to construct $\mathbf{u}$ and $\mathbf{v}$.

4.3.2 Perspective Views

For a perspective view, all the rays have the same origin, at the viewpoint; it is the directions that are different for each pixel. The image plane is no longer positioned at $\mathbf{e}$, but rather some distance d in front of $\mathbf{e}$; this distance is the *image plane distance*, often loosely called the *focal length*, because choosing d plays the same role as choosing focal length in a real camera. The direction of each ray is defined by the viewpoint and the position of the pixel on the image plane. This situation is illustrated in Figure 4.9, and the resulting procedure is similar to the

orthographic one:

compute u and v using (4.1)
ray.direction $\leftarrow -d\,\mathbf{w} + u\,\mathbf{u} + v\,\mathbf{v}$
ray.origin $\leftarrow \mathbf{e}$

As with parallel projection, oblique perspective views can be achieved by specifying the image plane normal separately from the projection direction, then replacing $-d\,\mathbf{w}$ with $d\mathbf{d}$ in the expression for the ray direction.

4.4 Ray-Object Intersection

Once we've generated a ray $\mathbf{e} + t\mathbf{d}$, we next need to find the first intersection with any object where $t > 0$. In practice, it turns out to be useful to solve a slightly more general problem: find the first intersection between the ray and a surface that occurs at a t in the interval $[t_0, t_1]$. The basic ray intersection is the case where $t_0 = 0$ and $t_1 = +\infty$. We solve this problem for both spheres and triangles. In the next section, multiple objects are discussed.

4.4.1 Ray-Sphere Intersection

Given a ray $\mathbf{p}(t) = \mathbf{e} + t\mathbf{d}$ and an implicit surface $f(\mathbf{p}) = 0$ (see Section 2.5.3), we'd like to know where they intersect. Intersection points occur when points on the ray satisfy the implicit equation, so the values of t we seek are those that solve the equation

$$f(\mathbf{p}(t)) = 0 \quad \text{or} \quad f(\mathbf{e} + t\mathbf{d}) = 0.$$

A sphere with center $\mathbf{c} = (x_c, y_c, z_c)$ and radius R can be represented by the implicit equation

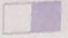
$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 - R^2 = 0.$$

We can write this same equation in vector form:

$$(\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) - R^2 = 0.$$

Any point $\mathbf{p}$ that satisfies this equation is on the sphere. If we plug points on the ray $\mathbf{p}(t) = \mathbf{e} + t\mathbf{d}$ into this equation, we get an equation in terms of t that is satisfied by the values of t that yield points on the sphere:

$$(\mathbf{e} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{e} + t\mathbf{d} - \mathbf{c}) - R^2 = 0.$$



Rearranging terms yields

$$(\mathbf{d}\cdot\mathbf{d})t^2 + 2\mathbf{d}\cdot(\mathbf{e}-\mathbf{c})t + (\mathbf{e}-\mathbf{c})\cdot(\mathbf{e}-\mathbf{c}) - R^2 = 0.$$

Here, everything is known except the parameter t, so this is a classic quadratic equation in t, meaning it has the form

$$At^2 + Bt + C = 0.$$

The solution to this equation is discussed in Section 2.2. The term under the square root sign in the quadratic solution, $B^2 - 4AC$, is called the *discriminant* and tells us how many real solutions there are. If the discriminant is negative, its square root is imaginary and the line and sphere do not intersect. If the discriminant is positive, there are two solutions: one solution where the ray enters the sphere and one where it leaves. If the discriminant is zero, the ray grazes the sphere, touching it at exactly one point. Plugging in the actual terms for the sphere and canceling a factor of two, we get

$$t = \frac{-\mathbf{d}\cdot(\mathbf{e}-\mathbf{c}) \pm \sqrt{(\mathbf{d}\cdot(\mathbf{e}-\mathbf{c}))^2 - (\mathbf{d}\cdot\mathbf{d})\left((\mathbf{e}-\mathbf{c})\cdot(\mathbf{e}-\mathbf{c}) - R^2\right)}}{(\mathbf{d}\cdot\mathbf{d})}.$$

In an actual implementation, you should first check the value of the discriminant before computing other terms. If the sphere is used only as a bounding object for more complex objects, then we need only determine whether we hit it; checking the discriminant suffices.

As discussed in Section 2.5.4, the normal vector at point $\mathbf{p}$ is given by the gradient $\mathbf{n} = 2(\mathbf{p}-\mathbf{c})$. The unit normal is $(\mathbf{p}-\mathbf{c})/R$.

4.4.2 Ray-Triangle Intersection

There are many algorithms for computing ray-triangle intersections. We will present the form that uses barycentric coordinates for the parametric plane containing the triangle, because it requires no long-term storage other than the vertices of the triangle (Snyder & Barr, 1987).

To intersect a ray with a parametric surface, we set up a system of equations where the Cartesian coordinates all match:

$$\left.\begin{aligned} x_e + tx_d &= f(u,v) \\ y_e + ty_d &= g(u,v) \\ z_e + tz_d &= h(u,v) \end{aligned}\right\} \quad \text{or,} \quad \mathbf{e} + t\mathbf{d} = \mathbf{f}(u,v).$$

Here, we have three equations and three unknowns (t, u, and v), so we can solve numerically for the unknowns. If we are lucky, we can solve for them analytically.

In the case where the parametric surface is a parametric plane, the parametric equation can be written in vector form as discussed in Section 2.7.2. If the vertices of the triangle are $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$, then the intersection will occur when

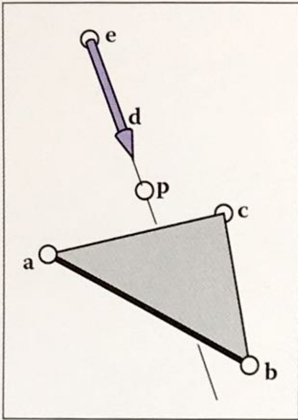


Figure 4.10. The ray hits the plane containing the triangle at point **p**.

$$\mathbf{e} + t\mathbf{d} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}), \tag{4.2}$$

for some t, β, and γ. The intersection $\mathbf{p}$ will be at $\mathbf{e} + t\mathbf{d}$ as shown in Figure 4.10. Again, from Section 2.7.2, we know the intersection is inside the triangle if and only if $\beta > 0$, $\gamma > 0$, and $\beta + \gamma < 1$. Otherwise, the ray has hit the plane outside the triangle, so it misses the triangle. If there are no solutions, either the triangle is degenerate or the ray is parallel to the plane containing the triangle.

To solve for t, β, and γ in Equation (4.2), we expand it from its vector form into the three equations for the three coordinates:

$$\begin{aligned} x_e + t x_d &= x_a + \beta(x_b - x_a) + \gamma(x_c - x_a), \\ y_e + t y_d &= y_a + \beta(y_b - y_a) + \gamma(y_c - y_a), \\ z_e + t z_d &= z_a + \beta(z_b - z_a) + \gamma(z_c - z_a). \end{aligned}$$

This can be rewritten as a standard linear system:

$$\begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_e \\ y_a - y_e \\ z_a - z_e \end{bmatrix}.$$

The fastest classic method to solve this 3×3 linear system is *Cramer's rule*. This gives us the solutions

$$\beta = \frac{\begin{vmatrix} x_a - x_e & x_a - x_c & x_d \\ y_a - y_e & y_a - y_c & y_d \\ z_a - z_e & z_a - z_c & z_d \end{vmatrix}}{|\mathbf{A}|},$$

$$\gamma = \frac{\begin{vmatrix} x_a - x_b & x_a - x_e & x_d \\ y_a - y_b & y_a - y_e & y_d \\ z_a - z_b & z_a - z_e & z_d \end{vmatrix}}{|\mathbf{A}|},$$

$$t = \frac{\begin{vmatrix} x_a - x_b & x_a - x_c & x_a - x_e \\ y_a - y_b & y_a - y_c & y_a - y_e \\ z_a - z_b & z_a - z_c & z_a - z_e \end{vmatrix}}{|\mathbf{A}|},$$

where the matrix $\mathbf{A}$ is

$$\mathbf{A} = \begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix},$$

and $|\mathbf{A}|$ denotes the determinant of $\mathbf{A}$. The 3×3 determinants have common subterms that can be exploited. Looking at the linear systems with dummy variables

$$\begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} j \\ k \\ l \end{bmatrix},$$

Cramer's rule gives us

$$\beta = \frac{j(ei - hf) + k(gf - di) + l(dh - eg)}{M},$$

$$\gamma = \frac{i(ak - jb) + h(jc - al) + g(bl - kc)}{M},$$

$$t = -\frac{f(ak - jb) + e(jc - al) + d(bl - kc)}{M},$$

where

$$M = a(ei - hf) + b(gf - di) + c(dh - eg).$$

We can reduce the number of operations by reusing numbers such as "*ei-minus-hf.*"

The algorithm for the ray-triangle intersection for which we need the linear solution can have some conditions for early termination. Thus, the function should look something like:

boolean raytri (ray $\mathbf{r}$, vector3 $\mathbf{a}$, vector3 $\mathbf{b}$, vector3 $\mathbf{c}$,
 interval $[t_0, t_1]$)
compute t
if $(t < t_0)$ or $(t > t_1)$ **then**
 return false
compute γ
if $(\gamma < 0)$ or $(\gamma > 1)$ **then**
 return false
compute β
if $(\beta < 0)$ or $(\beta > 1 - \gamma)$ **then**
 return false
return true

4.4.3 Ray-Polygon Intersection

Given a planar polygon with m vertices $\mathbf{p}_1$ through $\mathbf{p}_m$ and surface normal $\mathbf{n}$, we first compute the intersection points between the ray $\mathbf{e} + t\mathbf{d}$ and the plane containing the polygon with implicit equation

$$(\mathbf{p} - \mathbf{p}_1) \cdot \mathbf{n} = 0.$$

We do this by setting $\mathbf{p} = \mathbf{e} + t\mathbf{d}$ and solving for t to get

$$t = \frac{(\mathbf{p}_1 - \mathbf{e}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}.$$

This allows us to compute $\mathbf{p}$. If $\mathbf{p}$ is inside the polygon, then the ray hits it; otherwise, it does not.

We can answer the question of whether $\mathbf{p}$ is inside the polygon by projecting the point and polygon vertices to the xy plane and answering it there. The easiest way to do this is to send any 2D ray out from $\mathbf{p}$ and to count the number of intersections between that ray and the boundary of the polygon (Sutherland, Sproull, & Schumacker, 1974; Glassner, 1989). If the number of intersections is odd, then the point is inside the polygon; otherwise it is not. This is true because a ray that goes in must go out, thus creating a pair of intersections. Only a ray that starts inside will not create such a pair. To make computation simple, the 2D ray may as well propagate along the x-axis:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \end{bmatrix} + s \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

It is straightforward to compute the intersection of that ray with the edges such as (x_1, y_1, x_2, y_2) for $s \in (0, \infty)$.

A problem arises, however, for polygons whose projection into the xy plane is a line. To get around this, we can choose among the xy, yz, or zx planes for whichever is best. If we implement our points to allow an indexing operation, e.g., $\mathbf{p}(0) = x_p$ then this can be accomplished as follows:

if (abs(z_n) $>$ abs(x_n)) and (abs(z_n) $>$ abs(y_n)) **then**
  index0 = 0
  index1 = 1
else if (abs(y_n) $>$ abs (x_n)) **then**
  index0 = 0
  index1 = 2
else
  index0 = 1
  index1 = 2

Now, all computations can use $\mathbf{p}$(index0) rather than x_p, and so on.

Another approach to polygons, one that is often used in practice, is to replace them by several triangles.

4.4.4 Intersecting a Group of Objects

Of course, most interesting scenes consist of more than one object, and when we intersect a ray with the scene we must find only the closest intersection to the camera along the ray. A simple way to implement this is to think of a group of objects as itself being another type of object. To intersect a ray with a group, you simply intersect the ray with the objects in the group and return the intersection with the smallest t value. The following code tests for hits in the interval $t \in [t_0, t_1]$:

```
hit = false
for each object o in the group do
    if (o is hit at ray parameter t and t ∈ [t0, t1]) then
        hit = true
        hitobject = o
        t1 = t
return hit
```

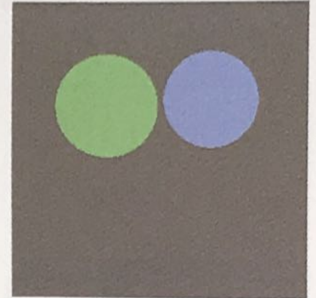


Figure 4.11. A simple scene rendered with only ray generation and surface intersection, but no shading; each pixel is just set to a fixed color depending on which object it hit.

4.5 Shading

Once the visible surface for a pixel is known, the pixel value is computed by evaluating a *shading model*. How this is done depends entirely on the application—methods range from very simple heuristics to elaborate numerical computations. In this chapter we describe the two most basic shading models; more advanced models are discussed in Chapter 10.

Most shading models, one way or another, are designed to capture the process of light reflection, whereby surfaces are illuminated by light sources and reflect part of the light to the camera. Simple shading models are defined in terms of illumination from a point light source. The important variables in light reflection are the light direction $\mathbf{l}$, which is a unit vector pointing toward the light source; the view direction $\mathbf{v}$, which is a unit vector pointing toward the eye or camera; the surface normal $\mathbf{n}$, which is a unit vector perpendicular to the surface at the point where reflection is taking place; and the characteristics of the surface—color, shininess, or other properties depending on the particular model.

4.5.1 Lambertian Shading

Illumination from real point sources falls off as distance squared, but that is often more trouble than it's worth in a simple renderer.

The simplest shading model is based on an observation made by Lambert in the 18th century: the amount of energy from a light source that falls on an area of surface depends on the angle of the surface to the light. A surface facing directly toward the light receives maximum illumination; a surface tangent to the light direction (or facing away from the light) receives no illumination; and in between the illumination is proportional to the cosine of the angle θ between the surface normal and the light source (Figure 4.12). This leads to the *Lambertian shading model*:

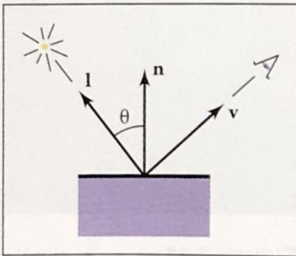


Figure 4.12. Geometry for Lambertian shading.

$$L = k_d \, I \max(0, \mathbf{n} \cdot \mathbf{l})$$

where L is the pixel color; k_d is the *diffuse coefficient*, or the surface color; and I is the intensity of the light source. Because $\mathbf{n}$ and $\mathbf{l}$ are unit vectors, we can use $\mathbf{n} \cdot \mathbf{l}$ as a convenient shorthand (both on paper and in code) for $\cos\theta$. This equation (as with the other shading equations in this section) applies separately to the three color channels, so the red component of the pixel value is the product of the red diffuse component, the red light source intensity, and the dot product; the same holds for green and blue.

When in doubt, make light sources neutral in color, with equal red, green, and blue intensities.

The vector $\mathbf{l}$ is computed by subtracting the intersection point of the ray and surface from the light source position. Don't forget that $\mathbf{v}$, $\mathbf{l}$, and $\mathbf{n}$ all must be unit vectors; failing to normalize these vectors is a very common error in shading computations.

4.5.2 Blinn-Phong Shading

Lambertian shading is *view independent*: the color of a surface does not depend on the direction from which you look. Many real surfaces show some degree of shininess, producing highlights, or *specular reflections*, that appear to move around as the viewpoint changes. Lambertian shading doesn't produce any highlights and leads to a very matte, chalky appearance, and many shading models add a *specular component* to Lambertian shading; the Lambertian part is then the *diffuse component*.

A very simple and widely used model for specular highlights was proposed by Phong (Phong, 1975) and later updated by Blinn (J. F. Blinn, 1976) to the form most commonly used today. The idea is to produce reflection that is at its brightest when $\mathbf{v}$ and $\mathbf{l}$ are symmetrically positioned across the surface normal, which is when mirror reflection would occur; the reflection then decreases smoothly as the vectors move away from a mirror configuration.

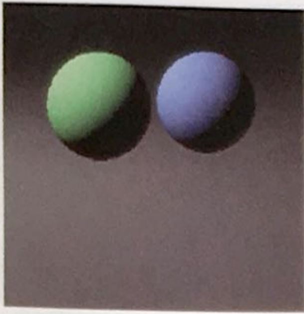


Figure 4.13. A simple scene rendered with diffuse shading from a single light source.

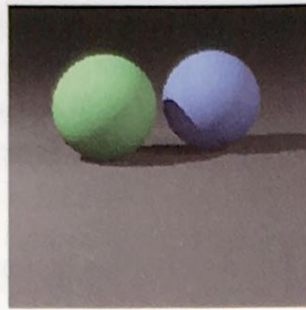


Figure 4.14. A simple scene rendered with diffuse shading and shadows (Section 4.7) from three light sources.

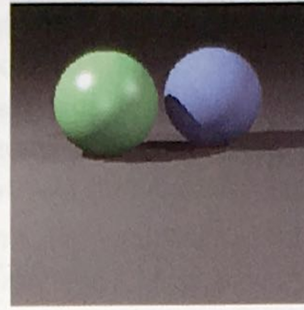


Figure 4.15. A simple scene rendered with diffuse shading (blue sphere), Blinn-Phong shading (green sphere), and shadows from three light sources.

We can tell how close we are to a mirror configuration by comparing the half vector $\mathbf{h}$ (the bisector of the angle between $\mathbf{v}$ and $\mathbf{l}$) to the surface normal (Figure 4.16). If the half vector is near the surface normal, the specular component should be bright; if it is far away it should be dim. This result is achieved by computing the dot product between $\mathbf{h}$ and $\mathbf{n}$ (remember they are unit vectors, so $\mathbf{n} \cdot \mathbf{h}$ reaches its maximum of 1 when the vectors are equal), then taking the result to a power $p > 1$ to make it decrease faster. The power, or *Phong exponent*, controls the apparent shininess of the surface. The half vector itself is easy to compute: since $\mathbf{v}$ and $\mathbf{l}$ are the same length, their sum is a vector that bisects the angle between them, which only needs to be normalized to produce $\mathbf{h}$.

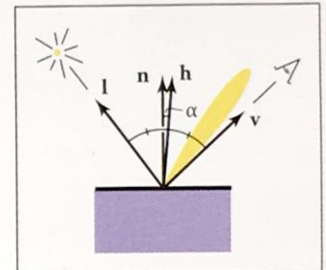


Figure 4.16. Geometry for Blinn-Phong shading.

Typical values of p:
10—"eggshell";
100—mildly shiny;
1000—really glossy;
10,000—nearly mirror-like.

When in doubt, make the specular color gray, with equal red, green, and blue values.

Putting this all together, the Blinn-Phong shading model is as follows:

$$\mathbf{h} = \frac{\mathbf{v} + \mathbf{l}}{\|\mathbf{v} + \mathbf{l}\|},$$

$$L = k_d\, I \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s\, I \max(0, \mathbf{n} \cdot \mathbf{h})^p,$$

where k_s is the *specular coefficient*, or the specular color, of the surface.

4.5.3 Ambient Shading

Surfaces that receive no illumination at all will be rendered as completely black, which is often not desirable. A crude but useful heuristic to avoid black shadows is to add a constant component to the shading model, one whose contribution to the pixel color depends only on the object hit, with no dependence on the surface geometry at all. This is known as ambient shading—it is as if surfaces were illuminated by "ambient" light that comes equally from everywhere. For

In the real world, surfaces that are not illuminated by light sources are illuminated by indirect reflections from other surfaces.

convenience in tuning the parameters, ambient shading is usually expressed as the product of a surface color with an ambient light color, so that ambient shading can be tuned for surfaces individually or for all surfaces together. Together with the rest of the Blinn-Phong model, ambient shading completes the full version of a simple and useful shading model:

$$L = k_a \, I_a + k_d \, I \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s \, I \max(0, \mathbf{n} \cdot \mathbf{h})^n, \tag{4.3}$$

When in doubt set the ambient color to be the same as the diffuse color.

where k_a is the surface's ambient coefficient, or "ambient color," and I_a is the ambient light intensity.

4.5.4 Multiple Point Lights

A very useful property of light is *superposition*—the effect caused by more than one light source is simply the sum of the effects of the light sources individually. For this reason, our simple shading model can easily be extended to handle N light sources:

$$L = k_a \, I_a + \sum_{i=1}^{N} \left[k_d \, I_i \max(0, \mathbf{n} \cdot \mathbf{l}_i) + k_s \, I_i \max(0, \mathbf{n} \cdot \mathbf{h}_i)^p \right], \tag{4.4}$$

where I_i, $\mathbf{l}_i$, and $\mathbf{h}_i$ are the intensity, direction, and half vector of the ith light source.

4.6 A Ray-Tracing Program

We now know how to generate a viewing ray for a given pixel, how to find the closest intersection with an object, and how to shade the resulting intersection. These are all the parts required for a program that produces shaded images with hidden surfaces removed.

for each pixel **do**
  compute viewing ray
  if (ray hits an object with $t \in [0, \infty)$) **then**
    Compute $\mathbf{n}$
    Evaluate shading model and set pixel to that color
  else
    set pixel color to background color

Here the statement "if ray hits an object . . ." can be implemented using the algorithm of Section 4.4.4.

In an actual implementation, the surface intersection routine needs to somehow return either a reference to the object that is hit, or at least its normal vector and shading-relevant material properties. This is often done by passing a record/structure with such information. In an object-oriented implementation, it is a good idea to have a class called something like *surface* with derived classes *triangle, sphere, group*, etc. Anything that a ray can intersect would be under that class. The ray-tracing program would then have one reference to a "surface" for the whole model, and new types of objects and efficiency structures can be added transparently.

4.6.1 Object-Oriented Design for a Ray-Tracing Program

As mentioned earlier, the key class hierarchy in a ray tracer are the geometric objects that make up the model. These should be subclasses of some geometric object class, and they should support a *hit* function (Kirk & Arvo, 1988). To avoid confusion from use of the word "object," *surface* is the class name often used. With such a class, you can create a ray tracer that has a general interface that assumes little about modeling primitives and debug it using only spheres. An important point is that anything that can be "hit" by a ray should be part of this class hierarchy, e.g., even a collection of surfaces should be considered a subclass of the surface class. This includes efficiency structures, such as bounding volume hierarchies; they can be hit by a ray, so they are in the class.

For example, the "abstract" or "base" class would specify the hit function as well as a bounding box function that will prove useful later:

```
class surface
    virtual bool hit(ray e + td, real t0, real t1, hit-record rec)
    virtual box bounding-box()
```

Here (t_0, t_1) is the interval on the ray where hits will be returned, and rec is a record that is passed by reference; it contains data such as the t at the intersection when hit returns true. The type box is a 3D "bounding box," that is two points that define an axis-aligned box that encloses the surface. For example, for a sphere, the function would be implemented by

```
box sphere::bounding-box()
    vector3 min = center − vector3(radius,radius,radius)
    vector3 max = center + vector3(radius,radius,radius)
    return box(min, max)
```

Another class that is useful is material. This allows you to abstract the material behavior and later add materials transparently. A simple way to link objects and materials is to add a pointer to a material in the surface class, although more programmable behavior might be desirable. A big question is what to do with textures; are they part of the material class or do they live outside of the material class? This will be discussed more in Chapter 11.

4.7 Shadows

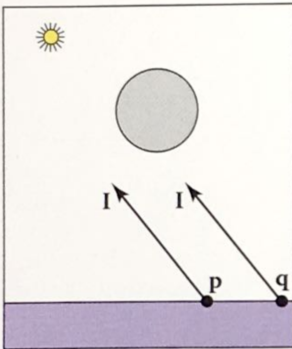


Figure 4.17. The point **p** is not in shadow, while the point **q** is in shadow.

Once you have a basic ray tracing program, shadows can be added very easily. Recall from Section 4.5 that light comes from some direction $\mathbf{l}$. If we imagine ourselves at a point $\mathbf{p}$ on a surface being shaded, the point is in shadow if we "look" in direction $\mathbf{l}$ and see an object. If there are no objects, then the light is not blocked.

This is shown in Figure 4.17, where the ray $\mathbf{p} + t\mathbf{l}$ does not hit any objects and is thus not in shadow. The point $\mathbf{q}$ is in shadow because the ray $\mathbf{q} + t\mathbf{l}$ does hit an object. The vector $\mathbf{l}$ is the same for both points because the light is "far" away. This assumption will later be relaxed. The rays that determine in or out of shadow are called *shadow rays* to distinguish them from viewing rays.

To get the algorithm for shading, we add an if statement to determine whether the point is in shadow. In a naive implementation, the shadow ray will check for $t \in [0, \infty)$, but because of numerical imprecision, this can result in an intersection with the surface on which $\mathbf{p}$ lies. Instead, the usual adjustment to avoid that problem is to test for $t \in [\epsilon, \infty)$ where ϵ is some small positive constant (Figure 4.18).

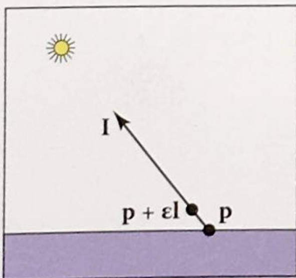


Figure 4.18. By testing in the interval starting at ϵ, we avoid numerical imprecision causing the ray to hit the surface $\mathbf{p}$ is on.

If we implement shadow rays for Phong lighting with Equation 4.3 then we have the following:

```
function raycolor( ray e + td, real t0, real t1 )
hit-record rec, srec
if (scene→hit(e + td, t0, t1, rec)) then
    p = e + (rec.t) d
    color c = rec.ka Ia
    if (not scene→hit(p + sl, ε, ∞, srec)) then
        vector3 h = normalized(normalized(l) + normalized(−d))
        c = c + rec.kd I max (0, rec.n · l) + (rec.ks) I (rec.n · h)^rec.p
    return c
else
    return background-color
```

Note that the ambient color is added whether $\mathbf{p}$ is in shadow or not. If there are multiple light sources, we can send a shadow ray before evaluating the shading model for each light. The code above assumes that $\mathbf{d}$ and $\mathbf{l}$ are not necessarily unit vectors. This is crucial for $\mathbf{d}$, in particular, if we wish to cleanly add *instancing* later (see Section 13.2).

4.8 Ideal Specular Reflection

It is straightforward to add *ideal specular* reflection, or *mirror reflection*, to a ray-tracing program. The key observation is shown in Figure 4.19 where a viewer looking from direction $\mathbf{e}$ sees what is in direction $\mathbf{r}$ as seen from the surface. The vector $\mathbf{r}$ is found using a variant of the Phong lighting reflection Equation (10.6). There are sign changes because the vector $\mathbf{d}$ points toward the surface in this case, so,

$$\mathbf{r} = \mathbf{d} - 2(\mathbf{d} \cdot \mathbf{n})\mathbf{n}. \tag{4.5}$$

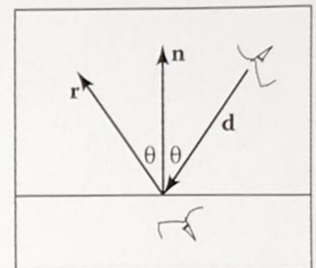


Figure 4.19. When looking into a perfect mirror, the viewer looking in direction **d** will see whatever the viewer "below" the surface would see in direction **r**.

In the real world, some energy is lost when the light reflects from the surface, and this loss can be different for different colors. For example, gold reflects yellow more efficiently than blue, so it shifts the colors of the objects it reflects. This can be implemented by adding a recursive call in *raycolor*:

$$\text{color } c = c + k_m \text{raycolor}(\mathbf{p} + s\mathbf{r}, \epsilon, \infty)$$

where k_m (for "mirror reflection") is the specular RGB color. We need to make sure we test for $s \in [\epsilon, \infty)$ for the same reason as we did with shadow rays; we don't want the reflection ray to hit the object that generates it.

The problem with the recursive call above is that it may never terminate. For example, if a ray starts inside a room, it will bounce forever. This can be fixed by adding a maximum recursion depth. The code will be more efficient if a reflection ray is generated only if k_m is not zero (black).

Figure 4.20. A simple scene rendered with diffuse and Blinn-Phong shading, shadows from three light sources, and specular reflection from the floor.

4.9 Historical Notes

Ray tracing was developed early in the history of computer graphics (Appel, 1968) but was not used much until sufficient compute power was available (Kay & Greenberg, 1979; Whitted, 1980).

Ray tracing has a lower asymptotic time complexity than basic object-order rendering (Snyder & Barr, 1987; Muuss, 1995; S. Parker et al., 1999; Wald, Slusallek, Benthin, & Wagner, 2001). Although it was traditionally thought of

as an offline method, real-time ray tracing implementations are becoming more and more common.

Frequently Asked Questions

- Why is there no perspective matrix in ray tracing?

The perspective matrix in a z-buffer exists so that we can turn the perspective projection into a parallel projection. This is not needed in ray tracing, because it is easy to do the perspective projection implicitly by fanning the rays out from the eye.

- Can ray tracing be made interactive?

For sufficiently small models and images, any modern PC is sufficiently powerful for ray tracing to be interactive. In practice, multiple CPUs with a shared frame buffer are required for a full-screen implementation. Computer power is increasing much faster than screen resolution, and it is just a matter of time before conventional PCs can ray trace complex scenes at screen resolution.

- Is ray tracing useful in a hardware graphics program?

Ray tracing is frequently used for *picking*. When the user clicks the mouse on a pixel in a 3D graphics program, the program needs to determine which object is visible within that pixel. Ray tracing is an ideal way to determine that.

Exercises

1. What are the ray parameters of the intersection points between ray $(1, 1, 1)+t(-1, -1, -1)$ and the sphere centered at the origin with radius 1? Note: this is a good debugging case.

2. What are the barycentric coordinates and ray parameter where the ray $(1, 1, 1) + t(-1, -1, -1)$ hits the triangle with vertices $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$? Note: this is a good debugging case.

3. Do a back of the envelope computation of the approximate time complexity of ray tracing on "nice" (non-adversarial) models. Split your analysis into the cases of preprocessing and computing the image, so that you can predict the behavior of ray tracing multiple frames for a static model.

the object to the root of the data structure. For example, consider the model of a ferry that has a car that can move freely on the deck of the ferry, and wheels that each move relative to the car as shown in Figure 12.21.

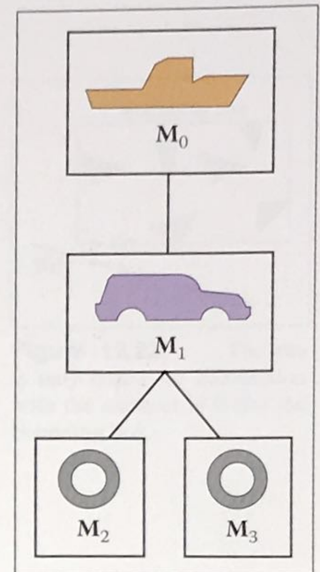


Figure 12.21. A ferry, a car on the ferry, and the wheels of the car (only two shown) are stored in a scene-graph.

As with the pendulum, each object should be transformed by the product of the matrices in the path from the root to the object:

- ferry transform using M_0;
- car body transform using M_0M_1;
- left wheel transform using $M_0M_1M_2$;
- left wheel transform using $M_0M_1M_3$.

An efficient implementation can be achieved using a *matrix stack*, a data structure supported by many APIs. A matrix stack is manipulated using *push* and *pop* operations that add and delete matrices from the right-hand side of a matrix product. For example, calling:

$$\text{push}(\mathbf{M}_0)$$
$$\text{push}(\mathbf{M}_1)$$
$$\text{push}(\mathbf{M}_2)$$

creates the active matrix $\mathbf{M} = \mathbf{M}_0\mathbf{M}_1\mathbf{M}_2$. A subsequent call to *pop()* strips the last matrix added so that the active matrix becomes $\mathbf{M} = \mathbf{M}_0\mathbf{M}_1$. Combining the matrix stack with a recursive traversal of a scene graph gives us:

```
function traverse(node)
push(M_local)
draw object using composite matrix from stack
traverse(left child)
traverse(right child)
pop()
```

There are many variations on scene graphs but all follow the basic idea above.

12.3 Spatial Data Structures

In many, if not all, graphics applications, the ability to quickly locate geometric objects in particular regions of space is important. Ray tracers need to find objects that intersect rays; interactive applications navigating an environment need to find the objects visible from any given viewpoint; games and physical simulations require detecting when and where objects collide. All these needs can be supported

by various *spatial data structures* designed to organize objects in space so they can be looked up efficiently.

In this section we will discuss examples of three general classes of spatial data structures. Structures that group objects together into a hierarchy are *object partitioning* schemes: objects are divided into disjoint groups, but the groups may end up overlapping in space. Structures that divide space into disjoint regions are *space partitioning* schemes: space is divided into separate partitions, but one object may have to intersect more than one partition. Space partitioning schemes can be regular, in which space is divided into uniformly shaped pieces, or irregular, in which space is divided adaptively into irregular pieces, with smaller pieces where there are more and smaller objects.

We will use ray tracing as the primary motivation while discussing these structures, though they can all also be used for view culling or collision detection. In Chapter 4, all objects were looped over while checking for intersections. For N objects, this is an $O(N)$ linear search and is thus slow for large scenes. Like most search problems, the ray-object intersection can be computed in sub-linear time using "divide and conquer" techniques, provided we can create an ordered data structure as a preprocess. There are many techniques to do this.

This section discusses three of these techniques in detail: bounding volume hierarchies (Rubin & Whitted, 1980; Whitted, 1980; Goldsmith & Salmon, 1987), uniform spatial subdivision (Cleary, Wyvill, Birtwistle, & Vatti, 1983; Fujimoto, Tanaka, & Iwata, 1986; Amanatides & Woo, 1987), and binary space partitioning (Glassner, 1984; Jansen, 1986; Havran, 2000). An example of the first two strategies is shown in Figure 12.22.

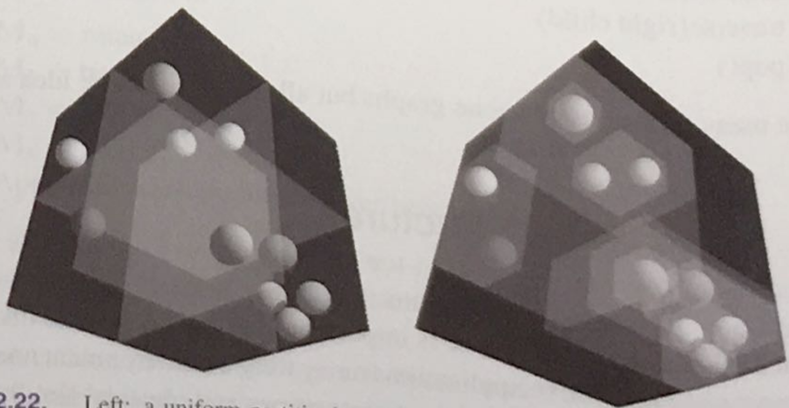


Figure 12.22. Left: a uniform partitioning of space. Right: adaptive bounding-box hierarchy. *Image courtesy David DeMarle.*

12.3.1 Bounding Boxes

A key operation in most intersection-acceleration schemes is computing the intersection of a ray with a bounding box (Figure 12.23). This differs from conventional intersection tests in that we do not need to know where the ray hits the box; we only need to know whether it hits the box.

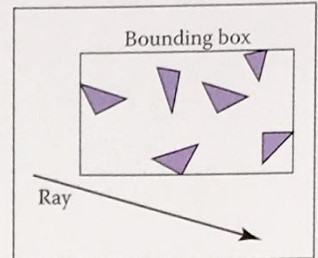


Figure 12.23. The ray is only tested for intersection with the surfaces if it hits the bounding box.

To build an algorithm for ray-box intersection, we begin by considering a 2D ray whose direction vector has positive x and y components. We can generalize this to arbitrary 3D rays later. The 2D bounding box is defined by two horizontal and two vertical lines:

$$x = x_{\text{min}},$$
$$x = x_{\text{max}},$$
$$y = y_{\text{min}},$$
$$y = y_{\text{max}}.$$

The points bounded by these lines can be described in interval notation:

$$(x, y) \in [x_{\text{min}}, x_{\text{max}}] \times [y_{\text{min}}, y_{\text{max}}].$$

As shown in Figure 12.24, the intersection test can be phrased in terms of these intervals. First, we compute the ray parameter where the ray hits the line $x = x_{\text{min}}$:

$$t_{\text{xmin}} = \frac{x_{\text{min}} - x_e}{x_d}.$$

We then make similar computations for t_{xmax}, t_{ymin}, and t_{ymax}. The ray hits the box if and only if the intervals $[t_{\text{xmin}}, t_{\text{xmax}}]$ and $[t_{\text{ymin}}, t_{\text{ymax}}]$ overlap, i.e., their intersection is nonempty. In pseudocode this algorithm is:

$t_{\text{xmin}} = (x_{\text{min}} - x_e)/x_d$
$t_{\text{xmax}} = (x_{\text{max}} - x_e)/x_d$
$t_{\text{ymin}} = (y_{\text{min}} - y_e)/y_d$
$t_{\text{ymax}} = (y_{\text{max}} - y_e)/y_d$
if $(t_{\text{xmin}} > t_{\text{ymax}})$ or $(t_{\text{ymin}} > t_{\text{xmax}})$ **then**
  return false
else
  return true

The if statement may seem non-obvious. To see the logic of it, note that there is no overlap if the first interval is either entirely to the right or entirely to the left of the second interval.

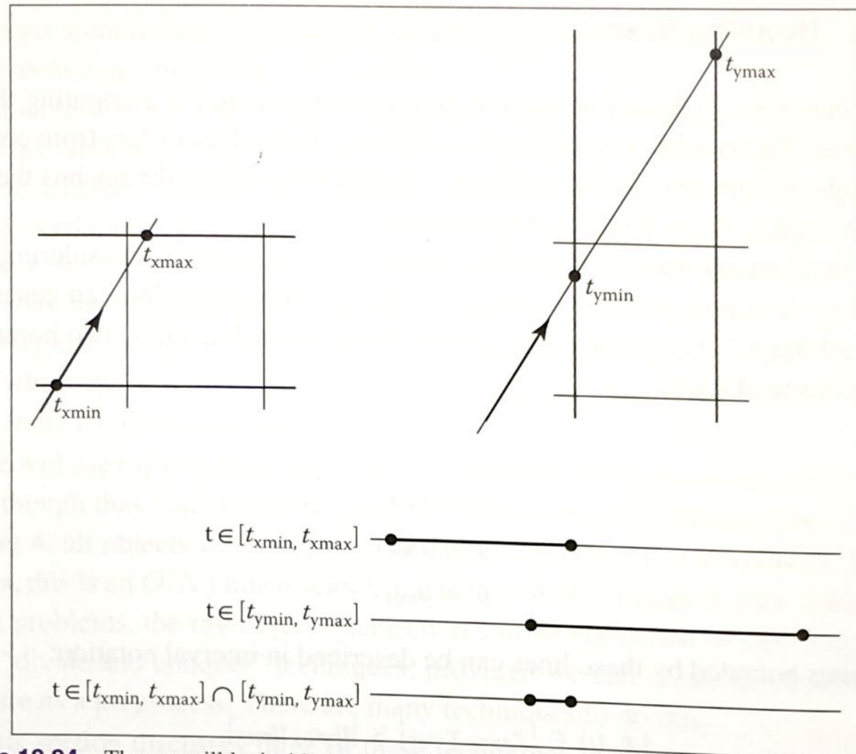


Figure 12.24. The ray will be inside the interval $x \in [x_{min}, x_{max}]$ for some interval in its parameter space $t \in [t_{xmin}, t_{xmax}]$. A similar interval exists for the y interval. The ray intersects the box if it is in both the x interval and y interval at the same time, i.e., the intersection of the two one-dimensional intervals is not empty.

The first thing we must address is the case when x_d or y_d is negative. If x_d is negative, then the ray will hit x_{max} before it hits x_{min}. Thus the code for computing t_{xmin} and t_{xmax} expands to:

if $(x_d \geq 0)$ **then**
  $t_{xmin} = (x_{min} - x_e)/x_d$
  $t_{xmax} = (x_{max} - x_e)/x_d$
else
  $t_{xmin} = (x_{max} - x_e)/x_d$
  $t_{xmax} = (x_{min} - x_e)/x_d$

A similar code expansion must be made for the y cases. A major concern is that horizontal and vertical rays have a zero value for y_d and x_d, respectively. This will cause divide by zero which may be a problem. However, before addressing this directly, we check whether IEEE floating point computation handles these cases gracefully for us. Recall from Section 1.5 the rules for divide by zero: for

any positive real number a,

$$+a/0 = +\infty;$$
$$-a/0 = -\infty.$$

Consider the case of a vertical ray where $x_d = 0$ and $y_d > 0$. We can then calculate

$$t_{\text{xmin}} = \frac{x_{\text{min}} - x_e}{0};$$
$$t_{\text{xmax}} = \frac{x_{\text{max}} - x_e}{0}.$$

There are three possibilities of interest:

1. $x_e \leq x_{\text{min}}$ (no hit);

2. $x_{\text{min}} < x_e < x_{\text{max}}$ (hit);

3. $x_{\text{max}} \leq x_e$ (no hit).

For the first case we have

$$t_{\text{xmin}} = \frac{\text{positive number}}{0};$$
$$t_{\text{xmax}} = \frac{\text{positive number}}{0}.$$

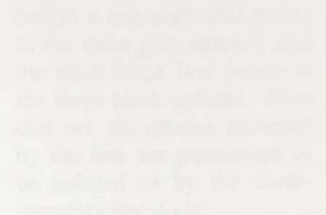
This yields the interval $(t_{\text{xmin}}, t_{\text{xmin}}) = (\infty, \infty)$. That interval will not overlap with any interval, so there will be no hit, as desired. For the second case, we have

$$t_{\text{xmin}} = \frac{\text{negative number}}{0};$$
$$t_{\text{xmax}} = \frac{\text{positive number}}{0}.$$

This yields the interval $(t_{\text{xmin}}, t_{\text{xmin}}) = (-\infty, \infty)$ which will overlap with all intervals and thus will yield a hit as desired. The third case results in the interval $(-\infty, -\infty)$ which yields no hit, as desired. Because these cases work as desired, we need no special checks for them. As is often the case, IEEE floating point conventions are our ally. However, there is still a problem with this approach.

Consider the code segment:

if $(x_d \geq 0)$ **then**
  $t_{\text{min}} = (x_{\text{min}} - x_e)/x_d$
  $t_{\text{max}} = (x_{\text{max}} - x_e)/x_d$



else
  $t_{\min} = (x_{\max} - x_e)/x_d$
  $t_{\max} = (x_{\min} - x_e)/x_d$

This code breaks down when $x_d = -0$. This can be overcome by testing on the reciprocal of x_d (A. Williams, Barrus, Morley, & Shirley, 2005):

$a = 1/x_d$
if $(a \geq 0)$ **then**
  $t_{\min} = a(x_{\min} - x_e)$
  $t_{\max} = a(x_{\max} - x_e)$
else
  $t_{\min} = a(x_{\max} - x_e)$
  $t_{\max} = a(x_{\min} - x_e)$

12.3.2 Hierarchical Bounding Boxes

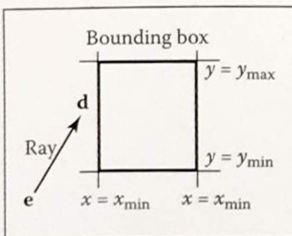


Figure 12.25. A 2D ray $\mathbf{e} + t\mathbf{d}$ is tested against a 2D bounding box.

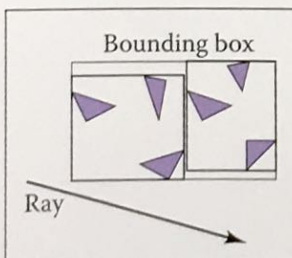


Figure 12.26. The bounding boxes can be nested by creating boxes around subsets of the model.

The basic idea of hierarchical bounding boxes can be seen by the common tactic of placing an axis-aligned 3D bounding box around all the objects as shown in Figure 12.25. Rays that hit the bounding box will actually be more expensive to compute than in a brute force search, because testing for intersection with the box is not free. However, rays that miss the box are cheaper than the brute force search. Such bounding boxes can be made hierarchical by partitioning the set of objects in a box and placing a box around each partition as shown in Figure 12.26. The data structure for the hierarchy shown in Figure 12.27 might be a tree with the large bounding box at the root and the two smaller bounding boxes as left and right subtrees. These would in turn each point to a list of three triangles. The intersection of a ray with this particular hard-coded tree would be:

if (ray hits root box) **then**
  if (ray hits left subtree box) **then**
    check three triangles for intersection
  if (ray intersects right subtree box) **then**
    check other three triangles for intersection
  if (an intersections returned from each subtree) **then**
    return the closest of the two hits
  else if (a intersection is returned from exactly one subtree) **then**
    return that intersection
  else
    return false
else
  return false

Some observations related to this algorithm are that there is no geometric ordering between the two subtrees, and there is no reason a ray might not hit both subtrees. Indeed, there is no reason that the two subtrees might not overlap.

A key point of such data hierarchies is that a box is guaranteed to bound all objects that are below it in the hierarchy, but they are *not* guaranteed to contain all objects that overlap it spatially, as shown in Figure 12.27. This makes this geometric search somewhat more complicated than a traditional binary search on strictly ordered one-dimensional data. The reader may note that several possible optimizations present themselves. We defer optimizations until we have a full hierarchical algorithm.

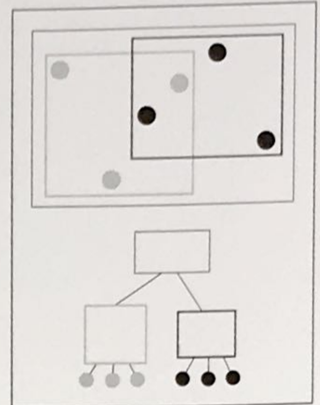


Figure 12.27. The gray box is a tree node that points to the three gray spheres, and the thick black box points to the three black spheres. Note that not all spheres enclosed by the box are guaranteed to be pointed to by the corresponding tree node.

If we restrict the tree to be binary and require that each node in the tree have a bounding box, then this traversal code extends naturally. Further, assume that all nodes are either leaves in the tree and contain a primitive, or that they contain one or two subtrees.

The `bvh-node` class should be of type surface, so it should implement `surface::hit`. The data it contains should be simple:

```
class bvh-node subclass of surface
  virtual bool hit(ray e + td, real t0, real t1, hit-record rec)
  virtual box bounding-box()
  surface-pointer left
  surface-pointer right
  box bbox
```

The traversal code can then be called recursively in an object-oriented style:

```
function bool bvh-node::hit(ray a + tb, real t0, real t1,
                             hit-record rec)
  if (bbox.hitbox(a + tb, t0, t1)) then
    hit-record lrec, rrec
    left-hit = (left ≠ NULL) and (left → hit(a + tb, t0, t1, lrec))
    right-hit = (right ≠ NULL) and (right → hit(a+tb, t0, t1, rrec))
    if (left-hit and right-hit) then
      if (lrec.t < rrec.t) then
        rec = lrec
      else
        rec = rrec
      return true
    else if (left-hit) then
      rec = lrec
      return true
    else if (right-hit) then
```

```
            rec = rrec
            return true
        else
            return false
    else
        return false
```

Note that because `left` and `right` point to surfaces rather than `bvh-nodes` specifically, we can let the virtual functions take care of distinguishing between internal and leaf nodes; the appropriate hit function will be called. Note that if the tree is built properly, we can eliminate the check for left being NULL. If we want to eliminate the check for right being NULL, we can replace NULL right pointers with a redundant pointer to left. This will end up checking left twice, but will eliminate the check throughout the tree. Whether that is worth it will depend on the details of tree construction.

There are many ways to build a tree for a bounding volume hierarchy. It is convenient to make the tree binary, roughly balanced, and to have the boxes of sibling subtrees not overlap too much. A heuristic to accomplish this is to sort the surfaces along an axis before dividing them into two sublists. If the axes are defined by an integer with $x = 0$, $y = 1$, and $z = 2$ we have:

```
function bvh-node::create(object-array A, int AXIS)
    N = A.length
    if (N= 1) then
        left = A[0]
        right = NULL
        bbox = bounding-box(A[0])
    else if (N= 2) then
        left-node = A[0]
        right-node = A[1]
        bbox = combine(bounding-box(A[0]), bounding-box(A[1]))
    else
        sort A by the object center along AXIS
        left= new bvh-node(A[0..N/2 − 1], (AXIS +1) mod 3)
        right = new bvh-node(A[N/2..N−1], (AXIS +1) mod 3)
        bbox = combine(left → bbox, right → bbox)
```

The quality of the tree can be improved by carefully choosing AXIS each time. One way to do this is to choose the axis such that the sum of the volumes of the bounding boxes of the two subtrees is minimized. This change compared to rotating through the axes will make little difference for scenes composed of isotopically distributed small objects, but it may help significantly in less well-behaved

scenes. This code can also be made more efficient by doing just a partition rather than a full sort.

Another, and probably better, way to build the tree is to have the subtrees contain about the same amount of space rather than the same number of objects. To do this we partition the list based on space:

```
function bvh-node::create(object-array A, int AXIS)
  N = A.length
  if (N = 1) then
    left = A[0]
    right = NULL
    bbox = bounding-box(A[0])
  else if (N = 2) then
    left = A[0]
    right = A[1]
    bbox = combine(bounding-box(A[0]), bounding-box(A[1]))
  else
    find the midpoint m of the bounding box of A along AXIS
    partition A into lists with lengths k and (N − k) surrounding m
    left = new bvh-node(A[0..k], (AXIS +1) mod 3)
    right = new bvh-node(A[k + 1..N − 1], (AXIS +1) mod 3)
    bbox = combine(left → bbox, right → bbox)
```

Although this results in an unbalanced tree, it allows for easy traversal of empty space and is cheaper to build because partitioning is cheaper than sorting.

12.3.3 Uniform Spatial Subdivision

Another strategy to reduce intersection tests is to divide space. This is fundamentally different from dividing objects as was done with hierarchical bounding volumes:

- In hierarchical bounding volumes, each object belongs to one of two sibling nodes, whereas a point in space may be inside both sibling nodes.
- In spatial subdivision, each point in space belongs to exactly one node, whereas objects may belong to many nodes.

In uniform spatial subdivision, the scene is partitioned into axis-aligned boxes. These boxes are all the same size, although they are not necessarily cubes. The ray traverses these boxes as shown in Figure 12.28. When an object is hit, the traversal ends.

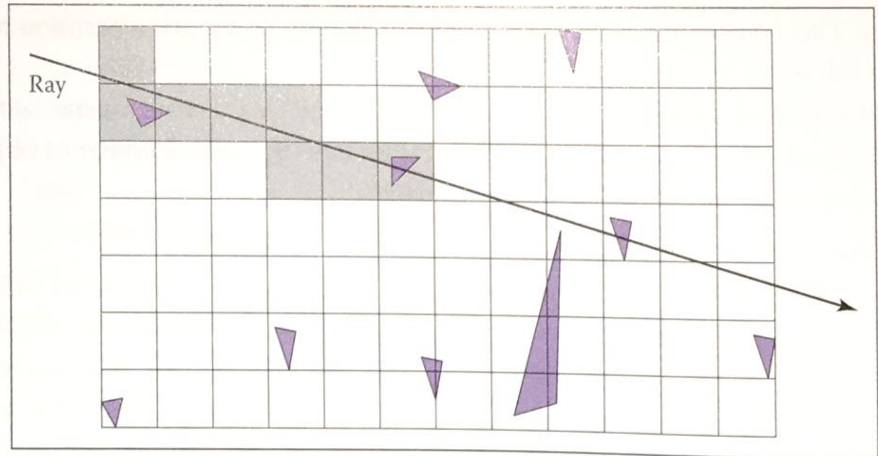


Figure 12.28. In uniform spatial subdivision, the ray is tracked forward through cells until an object in one of those cells is hit. In this example, only objects in the shaded cells are checked.

The grid itself should be a subclass of surface and should be implemented as a 3D array of pointers to surface. For empty cells these pointers are NULL. For cells with one object, the pointer points to that object. For cells with more than one object, the pointer can point to a list, another grid, or another data structure, such as a bounding volume hierarchy.

This traversal is done in an incremental fashion. The regularity comes from the way that a ray hits each set of parallel planes, as shown in Figure 12.29. To see how this traversal works, first consider the 2D case where the ray direction has positive x and y components and starts outside the grid. Assume the grid is bounded by points $(x_{\text{min}}, y_{\text{min}})$ and $(x_{\text{max}}, y_{\text{max}})$. The grid has $n_x \times n_y$ cells.

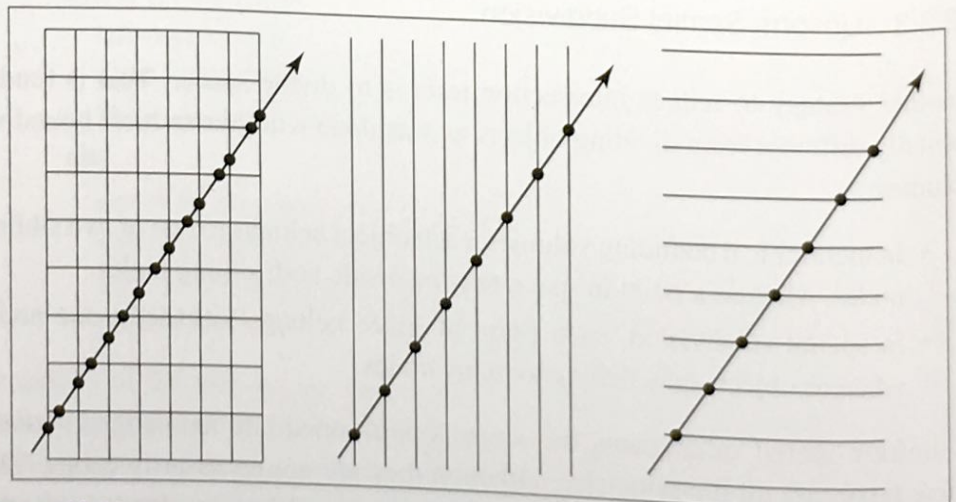


Figure 12.29. Although the pattern of cell hits seems irregular (left), the hits on sets of parallel planes are very even.

Our first order of business is to find the index (i, j) of the first cell hit by the ray $\mathbf{e} + t\mathbf{d}$. Then, we need to traverse the cells in an appropriate order. The key parts to this algorithm are finding the initial cell (i, j) and deciding whether to increment i or j (Figure 12.30). Note that when we check for an intersection with objects in a cell, we restrict the range of t to be within the cell (Figure 12.31). Most implementations make the 3D array of type "pointer to surface." To improve the locality of the traversal, the array can be tiled as discussed in Section 12.5.

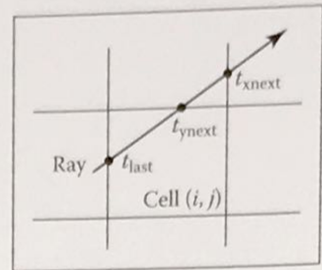


Figure 12.30. To decide whether we advance right or upward, we keep track of the intersections with the next vertical and horizontal boundary of the cell.

12.3.4 Axis-Aligned Binary Space Partitioning

We can also partition space in a hierarchical data structure such as a *binary space partitioning tree* (BSP tree). This is similar to the BSP tree used for visibility sorting in Section 12.4, but it's most common to use axis-aligned, rather than polygon-aligned, cutting planes for ray intersection.

A node in this structure contains a single cutting plane and a left and right subtree. Each subtree contains all the objects on one side of the cutting plane. Objects that pass through the plane are stored in in both subtrees. If we assume the cutting plane is parallel to the yz plane at $x = D$, then the node class is:

```
class bsp-node subclass of surface
   virtual bool hit(ray e + td, real t0, real t1, hit-record rec)
   virtual box bounding-box()
   surface-pointer left
   surface-pointer right
   real D
```

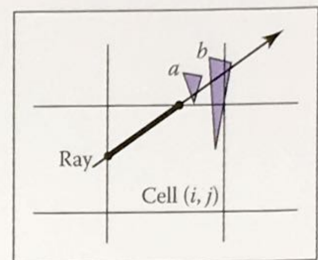


Figure 12.31. Only hits within the cell should be reported. Otherwise the case above would cause us to report hitting object *b* rather than object *a*.

We generalize this to y and z cutting planes later. The intersection code can then be called recursively in an object-oriented style. The code considers the four cases shown in Figure 12.32. For our purposes, the origin of these rays is a point at parameter t_0:

$$\mathbf{p} = \mathbf{a} + t_0\mathbf{b}.$$

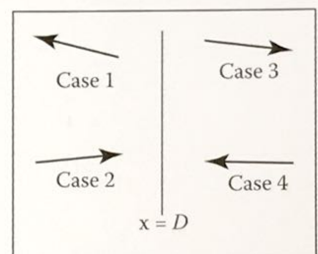


Figure 12.32. The four cases of how a ray relates to the BSP cutting plane $x = D$.

The four cases are:

1. The ray only interacts with the left subtree, and we need not test it for intersection with the cutting plane. It occurs for $x_p < D$ and $x_b < 0$.

2. The ray is tested against the left subtree, and if there are no hits, it is then tested against the right subtree. We need to find the ray parameter at $x = D$, so we can make sure we only test for intersections within the subtree. This case occurs for $x_p < D$ and $x_b > 0$.

3. This case is analogous to case 1 and occurs for $x_p > D$ and $x_b > 0$.

4. This case is analogous to case 2 and occurs for $x_p > D$ and $x_b < 0$.

The resulting traversal code handling these cases in order is:

```
function bool bsp-node::hit(ray a + tb, real t0, real t1,
                            hit-record rec)
  xp = xa + t0 xb
  if (xp < D) then
    if (xb < 0) then
      return (left ≠ NULL) and (left→hit(a + tb, t0, t1, rec))
    t = (D − xa)/xb
    if (t > t1) then
      return (left ≠ NULL) and (left→hit(a + tb, t0, t1, rec))
    if (left ≠ NULL) and (left→hit(a + tb, t0, t, rec)) then
      return true
    return (right ≠ NULL) and (right→hit(a + tb, t, t1, rec))
  else
    analogous code for cases 3 and 4
```

This is very clean code. However, to get it started, we need to hit some root object that includes a bounding box so we can initialize the traversal, t_0 and t_1. An issue we have to address is that the cutting plane may be along any axis. We can add an integer index *axis* to the *bsp-node* class. If we allow an indexing operator for points, this will result in some simple modifications to the code above, for example,

$$x_p = x_a + t_0 x_b$$

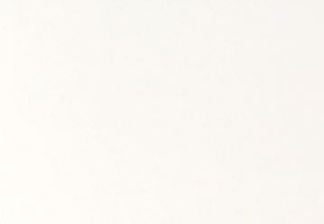
would become

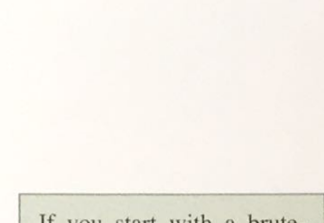
$$u_p = a[\text{axis}] + t_0 b[\text{axis}]$$

which will result in some additional array indexing, but will not generate more branches.

While the processing of a single bsp-node is faster than processing a bvh-node, the fact that a single surface may exist in more than one subtree means there are more nodes and, potentially, a higher memory use. How "well" the trees are built determines which is faster. Building the tree is similar to building the BVH tree. We can pick axes to split in a cycle, and we can split in half each time, or we can try to be more sophisticated in how we divide.

More Ray Tracing





A ray tracer is a great substrate on which to build all kinds of advanced rendering effects. Many effects that take significant work to fit into the object-order rasterization framework, including basics like the shadows and reflections already presented in Chapter 4, are simple and elegant in a ray tracer. In this chapter, we discuss some fancier techniques that can be used to ray-trace a wider variety of scenes and to include a wider variety of effects. Some extensions allow more general geometry: instancing and constructive solid geometry (CSG) are two ways to make models more complex with minimal complexity added to the program. Other extensions add to the range of materials we can handle: refraction through transparent materials, like glass and water, and glossy reflections on a variety of surfaces are essential for realism in many scenes.

This chapter also discusses the general framework of *distribution ray tracing* (Cook, Porter, & Carpenter, 1984), a powerful extension to the basic ray-tracing idea in which multiple random rays are sent through each pixel in an image to produce images with smooth edges and to simply and elegantly (if slowly) produce a wide range of effects from soft shadows to camera depth-of-field.

If you start with a brute-force ray intersection loop, you'll have ample time to implement an acceleration structure while you wait for images to render.

The price of the elegance of ray tracing is exacted in terms of computer time: most of these extensions will trace a very large number of rays for any nontrivial scene. Because of this, it's crucial to use the methods described in Chapter 12 to accelerate the tracing of rays.

13.1 Transparency and Refraction

In Chapter 4, we discussed the use of recursive ray tracing to compute specular, or mirror, reflection from surfaces. Another type of specular object is a *dielectric*—a transparent material that refracts light. Diamonds, glass, water, and air are dielectrics. Dielectrics also filter light; some glass filters out more red and blue light than green light, so the glass takes on a green tint. When a ray travels from a medium with refractive index n into one with a refractive index n_t, some of the light is transmitted, and it bends. This is shown for $n_t > n$ in Figure 13.1. Snell's Law tells us that

$$n \sin\theta = n_t \sin\phi.$$

Example values of n:
air: 1.00;
water: 1.33–1.34;
window glass: 1.51;
optical glass: 1.49–1.92;
diamond: 2.42.

Computing the sine of an angle between two vectors is usually not as convenient as computing the cosine, which is a simple dot product for the unit vectors such as we have here. Using the trigonometric identity $\sin^2\theta + \cos^2\theta = 1$, we can derive a refraction relationship for cosines:

$$\cos^2\phi = 1 - \frac{n^2\left(1 - \cos^2\theta\right)}{n_t^2}.$$

Note that if n and n_t are reversed, then so are θ and ϕ as shown on the right of Figure 13.1.

To convert $\sin\phi$ and $\cos\phi$ into a 3D vector, we can set up a 2D orthonormal basis in the plane of the surface normal, **n**, and the ray direction, **d**.

From Figure 13.2, we can see that **n** and **b** form an orthonormal basis for the plane of refraction. By definition, we can describe the direction of the transformed

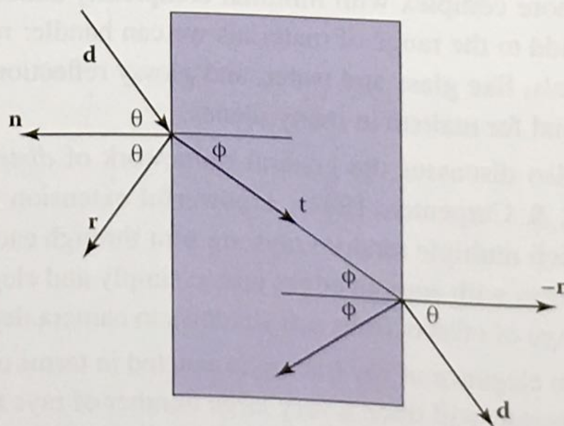


Figure 13.1. Snell's Law describes how the angle ϕ depends on the angle θ and the refractive indices of the object and the surrounding medium.

ray, $\mathbf{t}$, in terms of this basis:

$$\mathbf{t} = \sin\phi\mathbf{b} - \cos\phi\mathbf{n}.$$

Since we can describe $\mathbf{d}$ in the same basis, and $\mathbf{d}$ is known, we can solve for $\mathbf{b}$:

$$\mathbf{d} = \sin\theta\mathbf{b} - \cos\theta\mathbf{n},$$

$$\mathbf{b} = \frac{\mathbf{d} + \mathbf{n}\cos\theta}{\sin\theta}.$$

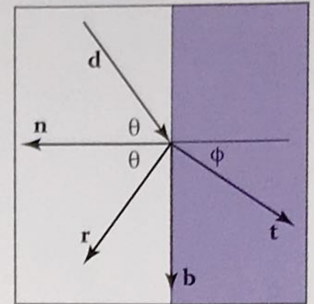


Figure 13.2. The vectors $\mathbf{n}$ and $\mathbf{b}$ form a 2D orthonormal basis that is parallel to the transmission vector $\mathbf{t}$.

This means that we can solve for $\mathbf{t}$ with known variables:

$$\mathbf{t} = \frac{n\left(\mathbf{d} + \mathbf{n}\cos\theta\right)}{n_t} - \mathbf{n}\cos\phi$$

$$= \frac{n\left(\mathbf{d} - \mathbf{n}(\mathbf{d}\cdot\mathbf{n})\right)}{n_t} - \mathbf{n}\sqrt{1 - \frac{n^2\left(1 - (\mathbf{d}\cdot\mathbf{n})^2\right)}{n_t^2}}.$$

Note that this equation works regardless of which of n and n_t is larger. An immediate question is, "What should you do if the number under the square root is negative?" In this case, there is no refracted ray and all of the energy is reflected. This is known as *total internal reflection*, and it is responsible for much of the rich appearance of glass objects.

The reflectivity of a dielectric varies with the incident angle according to the *Fresnel equations*. A nice way to implement something close to the Fresnel equations is to use the *Schlick approximation* (Schlick, 1994a),

$$R(\theta) = R_0 + (1 - R_0)\left(1 - \cos\theta\right)^5,$$

where R_0 is the reflectance at normal incidence:

$$R_0 = \left(\frac{n_t - 1}{n_t + 1}\right)^2.$$

Note that the $\cos\theta$ terms above are always for the angle in air (the larger of the internal and external angles relative to the normal).

For homogeneous impurities, as is found in typical colored glass, a light-carrying ray's intensity will be attenuated according to *Beer's Law*. As the ray travels through the medium it loses intensity according to $dI = -CI\,dx$, where dx is distance. Thus, $dI/dx = -CI$. We can solve this equation and get the exponential $I = k\exp(-Cx)$. The degree of attenuation is described by the RGB attenuation constant a, which is the amount of attenuation after one unit of distance. Putting in boundary conditions, we know that $I(0) = I_0$, and $I(1) =$

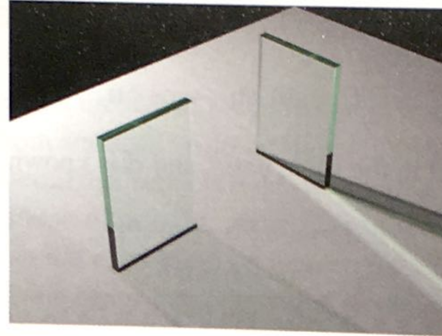


Figure 13.3. The color of the glass is affected by total internal reflection and Beer's Law. The amount of light transmitted and reflected is determined by the Fresnel equations. The complex lighting on the ground plane was computed using particle tracing as described in Chapter 23.

$aI(0)$. The former implies $I(x) = I_0 \exp(-Cx)$. The latter implies $I_0 a = I_0 \exp(-C)$, so $-C = \ln(a)$. Thus, the final formula is

$$I(s) = I(0)e^{\ln(a)s},$$

where $I(s)$ is the intensity of the beam at distance s from the interface. In practice, we reverse-engineer a by eye, because such data is rarely easy to find. The effect of Beer's Law can be seen in Figure 13.3, where the glass takes on a green tint.

To add transparent materials to our code, we need a way to determine when a ray is going "into" an object. The simplest way to do this is to assume that all objects are embedded in air with refractive index very close to 1.0, and that surface normals point "out" (toward the air). The code segment for rays and dielectrics with these assumptions is:

```
if (p is on a dielectric) then
  r = reflect(d, n )
  if (d · n < 0) then
    refract(d, n, n, t)
    c = −d · n
    k_r = k_g = k_b = 1
  else
    k_r = exp(−a_r t)
    k_g = exp(−a_g t)
    k_b = exp(−a_b t)
    if refract(d, −n, 1/n, t) then
      c = t · n
    else
      return k ∗ color(p + tr)
  R_0 = (n − 1)^2/(n + 1)^2
```

$R = R_0 + (1 - R_0)(1 - c)^5$
return $k(R \text{ color}(\mathbf{p} + t\mathbf{r}) + (1 - R) \text{ color}(\mathbf{p} + t\mathbf{t}))$

The code above assumes that the natural log has been folded into the constants (a_r, a_g, a_b). The *refract* function returns false if there is total internal reflection, and otherwise it fills in the last argument of the argument list.

13.2 Instancing

An elegant property of ray tracing is that it allows very natural *instancing*. The basic idea of instancing is to distort all points on an object by a transformation matrix before the object is displayed. For example, if we transform the unit circle (in 2D) by a scale factor $(2, 1)$ in x and y, respectively, then rotate it by 45°, and move one unit in the x-direction, the result is an ellipse with an eccentricity of 2 and a long axis along the $(x = -y)$-direction centered at $(0, 1)$ (Figure 13.4). The key thing that makes that entity an "instance" is that we store the circle and the composite transform matrix. Thus, the explicit construction of the ellipse is left as a future operation at render time.

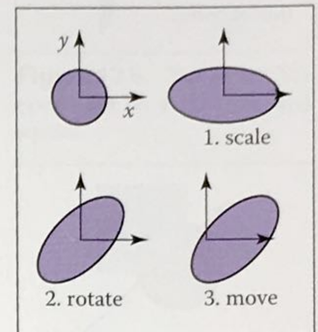


Figure 13.4. An instance of a circle with a series of three transforms is an ellipse.

The advantage of instancing in ray tracing is that we can choose the space in which to do intersection. If the base object is composed of a set of points, one of which is $\mathbf{p}$, then the transformed object is composed of that set of points transformed by matrix $\mathbf{M}$, where the example point is transformed to $\mathbf{Mp}$. If we have a ray $\mathbf{a} + t\mathbf{b}$ that we want to intersect with the transformed object, we can instead intersect an *inverse-transformed ray* with the untransformed object (Figure 13.5). There are two potential advantages to computing in the untransformed space (i.e., the right-hand side of Figure 13.5):

1. The untransformed object may have a simpler intersection routine, e.g., a sphere versus an ellipsoid.
2. Many transformed objects can share the same untransformed object thus reducing storage, e.g., a traffic jam of cars, where individual cars are just transforms of a few base (untransformed) models.

As discussed in Section 6.2.2, surface normal vectors transform differently. With this in mind and using the concepts illustrated in Figure 13.5, we can determine the intersection of a ray and an object transformed by matrix $\mathbf{M}$. If we create an instance class of type *surface*, we need to create a *hit* function:

instance::hit(ray $\mathbf{a} + t\mathbf{b}$, real t_0, real t_1, hit-record rec)
ray $\mathbf{r}' = \mathbf{M}^{-1}\mathbf{a} + t\mathbf{M}^{-1}\mathbf{b}$

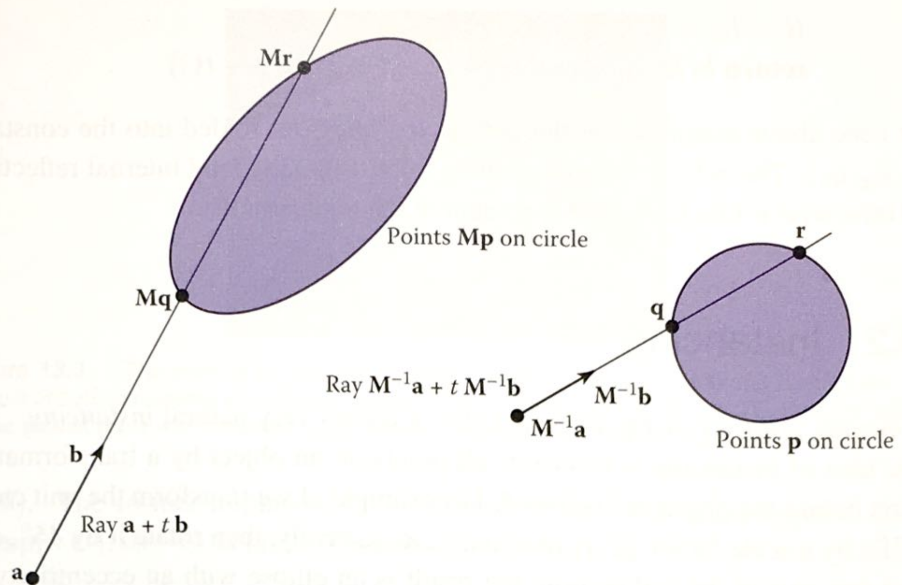


Figure 13.5. The ray intersection problem in the two spaces are just simple transforms of each other. The object is specified as a sphere plus matrix **M**. The ray is specified in the transformed (world) space by location **a** and direction **b**.

if (base-object→hit($\mathbf{r}'$, t_0, t_1, rec)) **then**
  rec.n $= (\mathrm{M}^{-1})^{\mathrm{T}}$rec.n
  return true
else
  return false

An elegant thing about this function is that the parameter rec.t does not need to be changed, because it is the same in either space. Also note that we need not compute or store the matrix M.

This brings up a very important point: the ray direction **b** must *not* be restricted to a unit-length vector, or none of the infrastructure above works. For this reason, it is useful not to restrict ray directions to unit vectors.

13.3 Constructive Solid Geometry

One nice thing about ray tracing is that any geometric primitive whose intersection with a 3D line can be computed can be seamlessly added to a ray tracer. It turns out to also be straightforward to add *constructive solid geometry* (CSG) to a ray

tracer (Roth, 1982). The basic idea of CSG is to use set operations to combine solid shapes. These basic operations are shown in Figure 13.6. The operations can be viewed as *set* operations. For example, we can consider C the set of all points in the circle and S the set of all points in the square. The intersection operation $C \cap S$ is the set of all points that are both members of C and S. The other operations are analogous.

Although one can do CSG directly on the model, if all that is desired is an image, we do not need to explicitly change the model. Instead, we perform the set operations directly on the rays as they interact with a model. To make this natural, we find all the intersections of a ray with a model rather than just the closest. For example, a ray $\mathbf{a} + t\mathbf{b}$ might hit a sphere at $t = 1$ and $t = 2$. In the context of CSG, we think of this as the ray being inside the sphere for $t \in [1, 2]$. We can compute these "inside intervals" for all of the surfaces and do set operations on those intervals (recall Section 2.1.2). This is illustrated in Figure 13.7, where the hit intervals are processed to indicate that there are two intervals inside the difference object. The first hit for $t > 0$ is what the ray actually intersects.

In practice, the CSG intersection routine must maintain a list of intervals. When the first hitpoint is determined, the material property and surface normal is that associated with the hitpoint. In addition, you must pay attention to precision issues because there is nothing to prevent the user from taking two objects that abut and taking an intersection. This can be made robust by eliminating any interval whose thickness is below a certain tolerance.

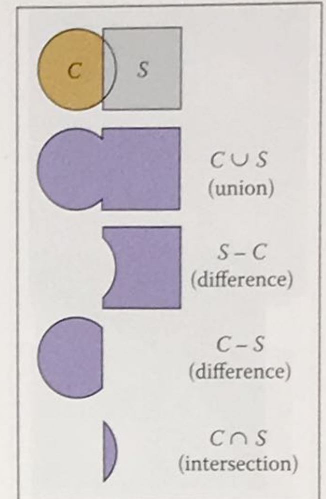


Figure 13.6. The basic CSG operations on a 2D circle and square.

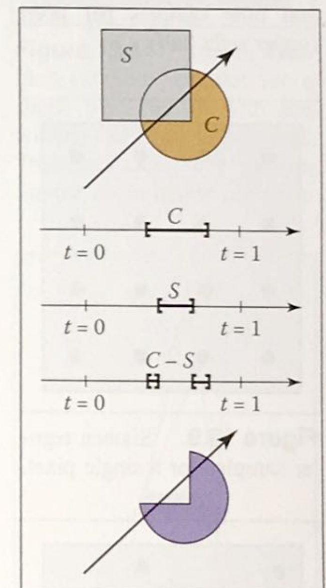


Figure 13.7. Intervals are processed to indicate how the ray hits the composite object.

13.4 Distribution Ray Tracing

For some applications, ray-traced images are just too "clean." This effect can be mitigated using *distribution ray tracing* (Cook et al., 1984). The conventionally ray-traced images look clean, because everything is crisp; the shadows are perfectly sharp, the reflections have no fuzziness, and everything is in perfect focus. Sometimes we would like to have the shadows be soft (as they are in real life), the reflections be fuzzy as with brushed metal, and the image have variable degrees of focus as in a photograph with a large aperture. While accomplishing these things from first principles is somewhat involved (as is developed in Chapter 23), we can get most of the visual impact with some fairly simple changes to the basic ray tracing algorithm. In addition, the framework gives us a relatively simple way to antialias (recall Section 8.3) the image.

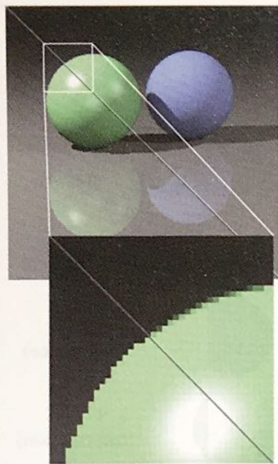


Figure 13.8. A simple scene rendered with one sample per pixel (lower left half) and nine samples per pixel (upper right half).

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$(i + (p + 0.5)/n, j + (q + 0.5)/n)$
  $c_{ij} = c/n^2$

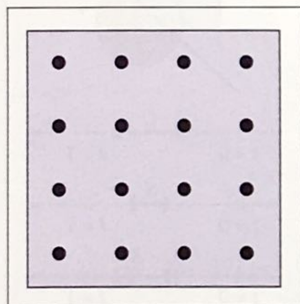


Figure 13.9. Sixteen regular samples for a single pixel.

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:

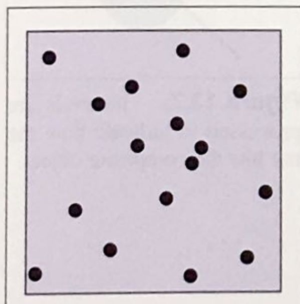


Figure 13.10. Sixteen random samples for a single pixel.

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).

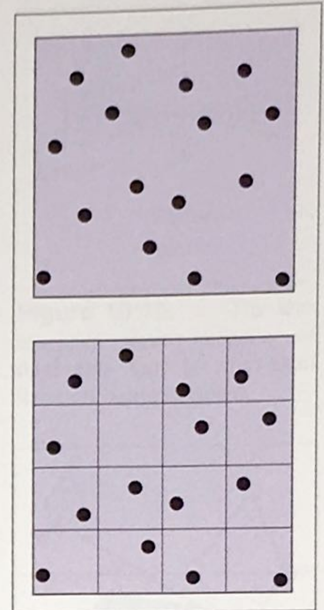


Figure 13.11. Sixteen stratified (jittered) samples for a single pixel shown with and without the bins highlighted. There is exactly one random sample taken within each bin.

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*.

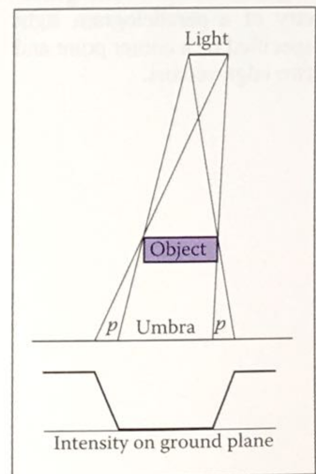


Figure 13.12. A soft shadow has a gradual transition from the unshadowed to shadowed region. The transition zone is the "penumbra" denoted by p in the figure.

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.

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

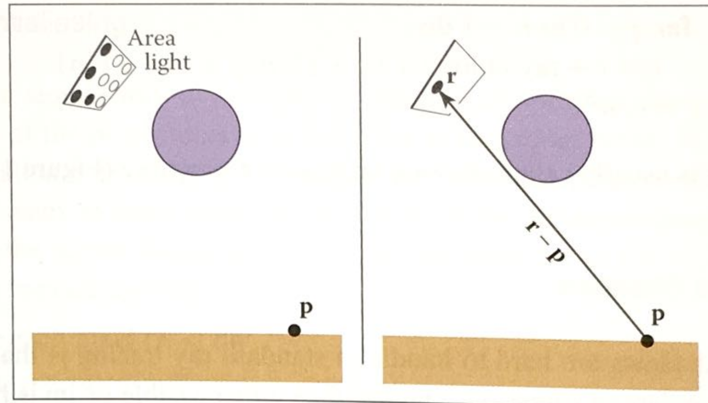


Figure 13.13. Left: an area light can be approximated by some number of point lights; four of the nine points are visible to **p** so it is in the penumbra. Right: a random point on the light is chosen for the shadow ray, and it has some chance of hitting the light or not.

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:

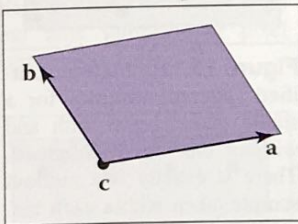


Figure 13.14. The geometry of a parallelogram light specified by a corner point and two edge vectors.

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.

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.

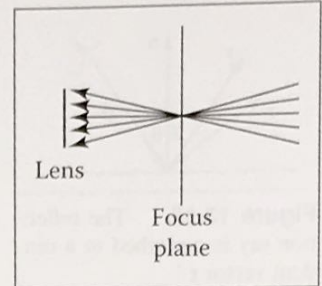


Figure 13.15. The lens averages over a cone of directions that hit the pixel location being sampled.

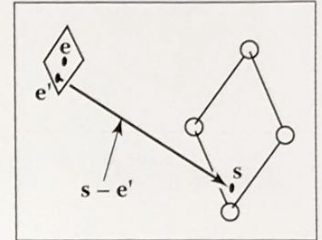


Figure 13.16. To create depth-of-field effects, the eye is randomly selected from a square region.

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.

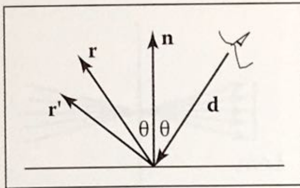


Figure 13.18. The reflection ray is perturbed to a random vector $\mathbf{r}'$.

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.