

Ray Tracing

Ray tracing is a method to produce realistic images; it determines visible surfaces in an image at the pixel level (Appel, 1968; Kay & Greenberg, 1979; Whitted, 1980). Unlike the z-buffer and BSP tree, ray tracing operates pixel-by-pixel rather than primitive-by-primitive. This tends to make ray tracing relatively slow for scenes with large objects in screen space. However, it has a variety of nice features which often make it the right choice for batch rendering and even for some interactive applications.

Ray tracing's primary benefit is that it is relatively straightforward to compute shadows and reflections. In addition, ray tracing is well suited to "walk-throughs" of extremely large models due to advanced ray tracing's low asymptotic time complexity which makes up for the required preprocessing of the model (Snyder & Barr, 1987; Muuss, 1995; Parker, Martin, et al., 1999; Wald, Slusallek, Benthin, & Wagner, 2001).

In an interactive 3D program implemented in a conventional z-buffer environment, it is often useful to be able to select an object using a mouse. The mouse is clicked in pixel (i, j) and the "picked" object is whatever object is "seen" through that pixel. If the rasterization process includes an object identification buffer, this is just a matter of looking up the value in pixel (i, j) of that buffer. However, if that buffer is not available, we can solve the problem of which object is visible via brute force geometrical computation using a "ray intersection test." In this way, ray tracing is useful also to programmers who use only standard graphics APIs.

This chapter also discusses *distribution ray tracing* (Cook, Porter, & Carpenter, 1984), where multiple random rays are sent through each pixel in an image to simultaneously solve the antialiasing, soft shadow, fuzzy reflection, and depth-of-field problems.

10.1 The Basic Ray-Tracing Algorithm

The simplest use of ray tracing is to produce images similar to those produced by the z-buffer and BSP-tree algorithms. Fundamentally, those methods make sure the appropriate object is “seen” through each pixel, and that the pixel color is shaded based on that object’s material properties, the surface normal seen through that pixel, and the light geometry.

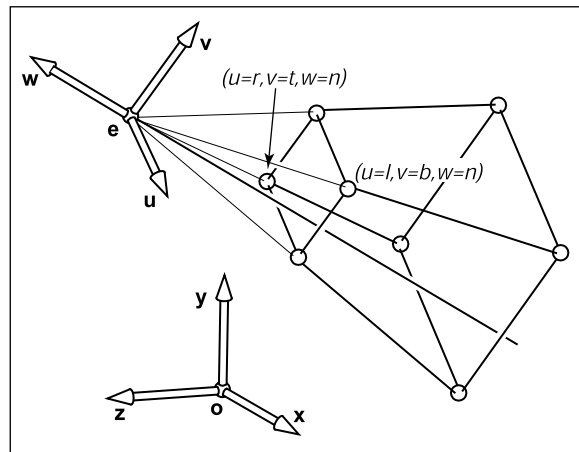


Figure 10.1. The 3D window we look through is the same as in Chapter 7. The borders of the window have simple coordinates in the uvw -coordinate system with respect to origin e .

Figure 10.1 shows the basic viewing geometry for ray tracing, which is the same as we saw earlier in Chapter 7. The geometry is aligned to a uvw coordinate system with the origin at the eye location e . The key idea in ray tracing is to identify locations on the view plane at $w = n$ that correspond to pixel centers, as shown in Figure 10.2. A “ray,” really just a directed 3D line, is then sent from e to that point. We then “gaze” in the direction of the ray to see the first object seen in that direction. This is shown in Figure 10.3, where the ray intersects two triangles, but only the first triangle hit, T_2 , is returned.

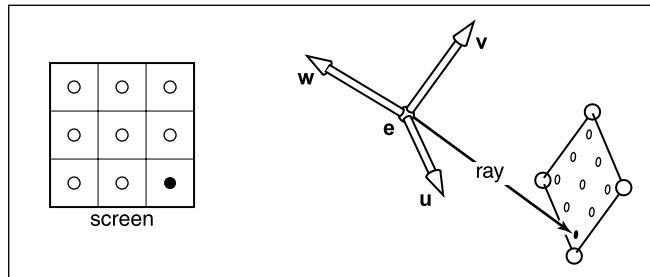


Figure 10.2. The sample points on the screen are mapped to a similar array on the 3D window. A viewing ray is sent to each of these locations.

The structure of the basic ray tracing program is:

 Compute u , v , w basis vectors

for each pixel **do**

 compute viewing ray

 find first object hit by ray and its surface normal n

 set pixel color to value based on material, light, and n

The pixel color can be computed using the shading equations of the last chapter.

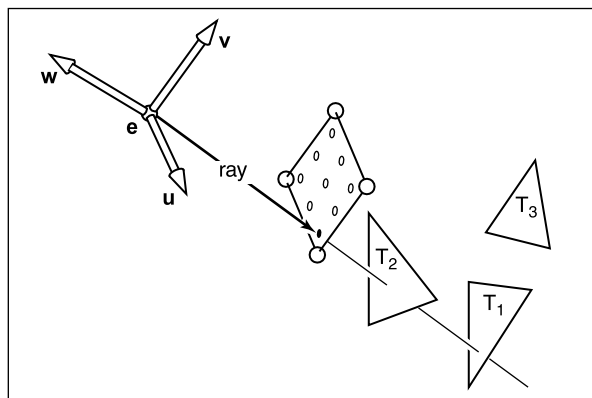


Figure 10.3. The ray is “traced” into the scene and the first object hit is the one seen through the pixel. In this case, the triangle T_2 is returned.

10.2 Computing Viewing Rays

First we need to determine 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

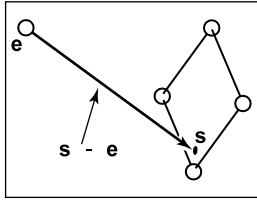


Figure 10.4. The ray from the eye to a point on the screen.

this. As discussed in Section 2.8.1, the 3D parametric line from the eye \mathbf{e} to a point \mathbf{s} on the screen (see Figure 10.4) is given by

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

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} . Note that $\mathbf{p}(0) = \mathbf{e}$, and $\mathbf{p}(1) = \mathbf{s}$. Also note that for positive t , if $t_1 < t_2$, 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. Note that we are overloading the variable t here which is also used for the top of the screen’s v -coordinate.

To compute a viewing ray, we need to know \mathbf{e} (which is given) and \mathbf{s} . Finding \mathbf{s} may look somewhat difficult. In fact, it is relatively straightforward using the same transform machinery we used for viewing in the context of projecting lines and triangles.

First, we find the coordinates of \mathbf{s} in the uvw -coordinate system with origin \mathbf{e} . For all points on the screen, $w_s = n$ as shown in Figure 10.2. The uv -coordinates are found by the windowing transform that takes $[-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5]$ to $[l, r] \times [b, t]$:

$$u_s = l + (r - l) \frac{i + 0.5}{n_x},$$

$$v_s = b + (t - b) \frac{j + 0.5}{n_y},$$

where (i, j) are the pixel indices. This gives us \mathbf{s} in uvw -coordinates. By definition, we can convert to canonical coordinates:

$$\mathbf{s} = \mathbf{e} + u_s \mathbf{u} + v_s \mathbf{v} + w_s \mathbf{w}. \quad (10.1)$$

Alternatively, we could use the matrix form (Equation 6.8):

$$\begin{bmatrix} x_s \\ y_s \\ z_s \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & x_e \\ 0 & 1 & 0 & y_e \\ 0 & 0 & 1 & z_e \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & x_v & x_w & 0 \\ y_u & y_v & y_w & 0 \\ z_u & z_v & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_s \\ v_s \\ w_s \\ 1 \end{bmatrix}, \quad (10.2)$$

which is just the matrix form of Equation 10.1. We can compose this with the windowing transform in matrix form if we wished, but this is probably not worth doing unless you like the matrix form of equations better.



10.3 Ray-Object Intersection

Given a ray $\mathbf{e} + t\mathbf{d}$, we want to find the first intersection with any object where $t > 0$. It will later prove useful to solve a slightly more general problem of finding the first intersection in the interval $[t_0, t_1]$, and using $[0, \infty)$ for viewing rays. We solve this for both spheres and triangles in this section. In the next section, multiple objects are discussed.

10.3.1 Ray-Sphere Intersection

Given a ray $\mathbf{p}(t) = \mathbf{e} + t\mathbf{d}$ and an implicit surface $f(\mathbf{p}) = 0$, we'd like to know where they intersect. The intersection points occur when points on the ray satisfy the implicit equation

$$f(\mathbf{p}(t)) = 0.$$

This is just

$$f(\mathbf{e} + t\mathbf{d}) = 0.$$

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 can solve for the values of t on the ray 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 there are no intersections between the sphere and the line. 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 eliminating the common factors of two, we get

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

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

10.3.2 Ray-Triangle Intersection

There are many algorithms for computing ray-triangle intersections. We will use 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:

$$\begin{aligned}x_e + tx_d &= f(u, v), \\y_e + ty_d &= g(u, v), \\z_e + tz_d &= h(u, v).\end{aligned}$$

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 2.11.2. If the vertices of the triangle are \mathbf{a} , \mathbf{b} and \mathbf{c} , then the intersection will occur when

$$\mathbf{e} + t\mathbf{d} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}). \quad (10.3)$$

The hitpoint \mathbf{p} will be at $\mathbf{e} + t\mathbf{d}$ as shown in Figure 10.5. Again, from Section 2.11.2, we know the hitpoint is in the triangle if and only if $\beta > 0$, $\gamma > 0$,

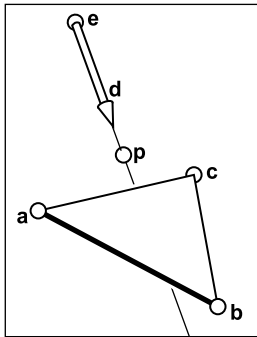


Figure 10.5. The ray hits the plane containing the triangle at point \mathbf{p} .

and $\beta + \gamma < 1$. Otherwise, it hits the plane outside 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 10.3, we expand it from its vector form into the three equations for the three coordinates:

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

This can be rewritten as a standard linear equation:

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

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

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

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 sub-terms 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 r, vector3 a, vector3 b, vector3 c, interval [t0, t1])
  compute t
  if (t < t0) or (t > t1) then
    return false
  compute γ
  if (γ < 0) or (γ > 1) then
    return false
  compute β
  if (β < 0) or (β > 1 - γ) then
    return false
  return true

```

10.3.3 Ray-Polygon Intersection

Given a 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, and 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 et al., 1974; Glassner, 1989). If the number of intersections is odd, then the point is inside the polygon, and 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( $x_n$ )) 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 $\mathbf{p}(\text{index0})$ rather than x_p , and so on.

10.4 A Ray-Tracing Program

We now know how to generate a viewing ray for a given pixel and how to find the intersection with one object. This can be easily extended to a program that produces images similar to the z-buffer or BSP-tree codes of earlier chapters:

```

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

```

Here the statement “if ray hits an object...” can be implemented as a function that tests for hits in the interval $t \in [t_0, t_1]$:

```

hit = false
for each object o do
  if (object is hit at ray parameter  $t$  and  $t \in [t_0, t_1]$ ) then
    hit = true
    hitobject = o
     $t_1 = t$ 
return hit

```

In an actual implementation, you will need to somehow return either a reference to the object that is hit or at least its normal vector and 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*, *surface-list*, 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.

10.4.1 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 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.

10.5 Shadows

Once you have a basic ray tracing program, shadows can be added very easily. Recall from Chapter 9 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 10.6, 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 inter-

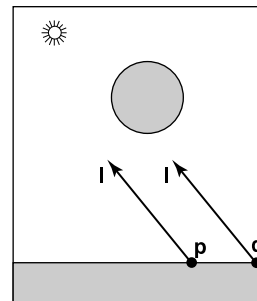


Figure 10.6. The point \mathbf{p} is not in shadow while the point \mathbf{q} is in shadow.

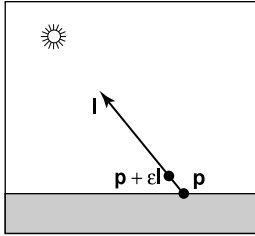


Figure 10.7. By testing in the interval starting at ϵ , we avoid numerical imprecision causing the ray to hit the surface \mathbf{p} is on.

section 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 10.7).

If we implement shadow rays for Phong lighting with Equation 9.9 then we have:

```

function raycolor( ray  $\mathbf{e} + t\mathbf{d}$ , real  $t_0$ , real  $t_1$  )
hit-record rec, srec
if (scene→hit( $\mathbf{e} + t\mathbf{d}$ ,  $t_0$ ,  $t_1$ , rec)) then
   $\mathbf{p} = \mathbf{e} + \text{rec}.t\mathbf{d}$ 
  color  $c = \text{rec}.c_r \text{rec}.c_a$ 
  if (not scene→hit( $\mathbf{p} + s\mathbf{l}$ ,  $\epsilon$ ,  $\infty$ , srec)) then
    vector3  $\mathbf{h} = \text{normalized}(\text{normalized}(\mathbf{l}) + \text{normalized}(-\mathbf{d}))$ 
     $c = c + \text{rec}.c_r c_l \max(0, \text{rec}.\mathbf{n} \cdot \mathbf{l}) + c_l \text{rec}.c_p (\mathbf{h} \cdot \text{rec}.\mathbf{n})^{\text{rec}.p}$ 
  return  $c$ 
else
  return background-color

```

Note that the ambient color is added in either case. If there are multiple light sources, we can send a shadow ray and evaluate the diffuse/phong terms 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.

10.6 Specular Reflection

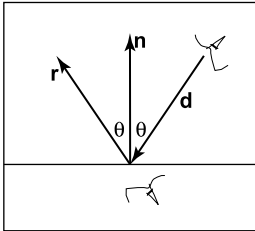


Figure 10.8. When looking into a perfect mirror, the viewer looking in direction \mathbf{d} will see whatever the viewer “below” the surface would see in direction \mathbf{r} .

It is straightforward to add *specular* reflection to a ray-tracing program. The key observation is shown in Figure 10.8 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 9.6. There are sign changes because the vector \mathbf{d} points toward the surface in this case, so,

$$\mathbf{r} = \mathbf{d} + 2(\mathbf{d} \cdot \mathbf{n})\mathbf{n}, \quad (10.4)$$

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*:

```

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

```

where c_s 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 c_s is not zero (black).

10.7 Refraction

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 10.9. Snell's law tells us that

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

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:

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

Note that if n and n_t are reversed, then so are θ and ϕ as shown on the right of Figure 10.9.

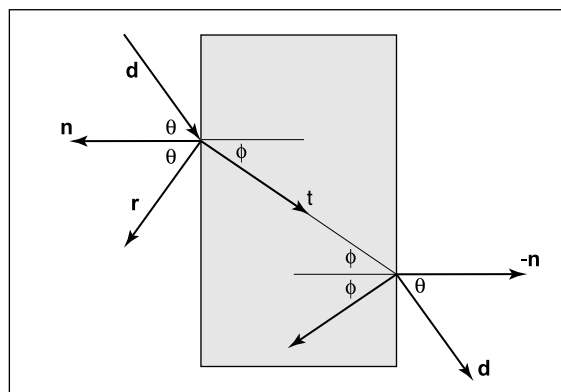


Figure 10.9. Snell's Law describes how the angle ϕ depends on the angle θ and the refractive indices of the object and the surrounding medium.

To convert $\sin \phi$ and $\cos \phi$ into a 3D vector, we can set up a 2D orthonormal basis in the plane of \mathbf{n} and \mathbf{d} .

From Figure 10.10, we can see that \mathbf{n} and \mathbf{b} form an orthonormal basis for the plane of refraction. By definition, we can describe \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} :

$$\begin{aligned} \mathbf{d} &= \sin \theta \mathbf{b} - \cos \theta \mathbf{n}, \\ \mathbf{b} &= \frac{\mathbf{d} + \mathbf{n} \cos \theta}{\sin \theta}. \end{aligned}$$

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

$$\begin{aligned} \mathbf{t} &= \frac{n(\mathbf{d} + \mathbf{n} \cos \theta)}{n_t} - \mathbf{n} \cos \phi \\ &= \frac{n(\mathbf{d} - \mathbf{n}(\mathbf{d} \cdot \mathbf{n}))}{n_t} - \mathbf{n} \sqrt{1 - \frac{n^2(1 - (\mathbf{d} \cdot \mathbf{n})^2)}{n_t^2}}. \end{aligned}$$

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),

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

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 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 $I = k \exp(-Cx) + k'$. The degree of attenuation is described by the RGB attenuation

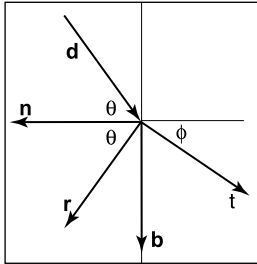


Figure 10.10. The vectors \mathbf{n} and \mathbf{b} form a 2D orthonormal basis that is parallel to the transmission vector \mathbf{t} .

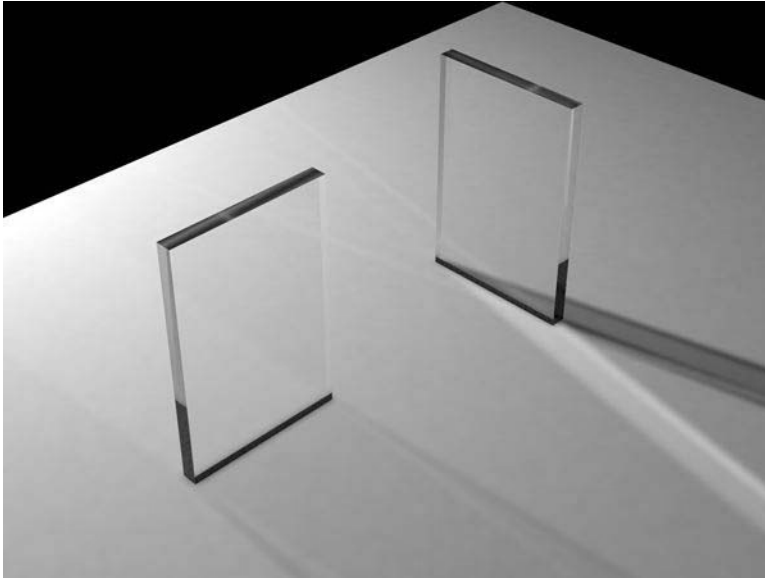


Figure 10.11. The color of the glass is affected by total internal reflection and Beer's Law. The amount of light transmitted and reflected is determined by the Fresnel equations. The complex lighting on the ground plane was computed using particle tracing as described in Chapter 23. (See also Plate VI.)

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 $I(1) = aI(0)$. 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 10.11, 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
kr = kg = kb = 1
else
kr = exp(-art)
kg = exp(-agt)
kb = exp(-abt)
if refract(d, -n, 1/n, t) then
c = t · n
else
return k * color(p + tr)
R0 = (n - 1)2 / (n + 1)2
R = R0 + (1 - R0)(1 - c)5
return k(R color(p + tr) + (1 - R) color(p + tt))

```

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.

10.8 Instancing

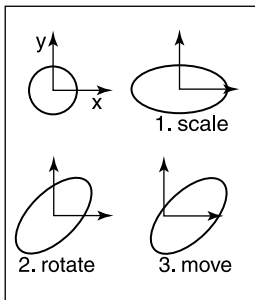


Figure 10.12. An instance of a circle with a series of three transforms is an ellipse.

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 10.12). 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 procedure operation at render time.

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 $\mathbf{M}\mathbf{p}$. If we have a ray $\mathbf{a} + t\mathbf{b}$ which we want to intersect with the transformed object, we can instead intersect an *inverse-transformed ray* with the untransformed object (Figure 10.13). There are two potential advantages to computing in the untransformed space (i.e., the right-hand side of Figure 10.13):

1. the untransformed object may have a simpler intersection routine, e.g., a sphere versus an ellipsoid;

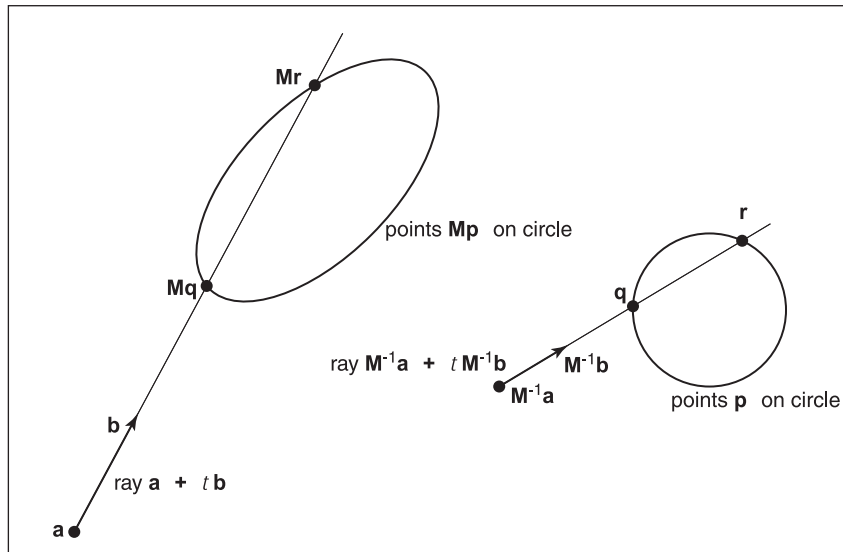


Figure 10.13. The ray intersection problem in the two spaces are just simple transforms of each other. The object is specified as a sphere plus matrix M . The ray is specified in the transformed (world) space by location \mathbf{a} and direction \mathbf{b} .

- 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 10.13, we can determine the intersection of a ray and an object transformed by matrix 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  $\mathbf{r}' = M^{-1}\mathbf{a} + tM^{-1}\mathbf{b}$ 
if (base-object→hit( $\mathbf{r}'$ ,  $t_0$ ,  $t_1$ , rec)) then
  rec. $\mathbf{n} = (M^{-1})^T$ rec. $\mathbf{n}$ 
  return true
else
  return false
```

An elegant thing about this function is that the parameter $\text{rec}.t$ 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 \mathbf{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.

For the purpose of solid texturing, you may want to record the local coordinates of the hitpoint and return this in the hit-record. This is just ray \mathbf{r}' advanced by parameter rec.t .

To implement the bounding-box function of class instance, we can just take the eight corners of the bounding box of the base object and transform all of them by \mathbf{M} , and then take the bounding box of those eight points. That will not necessarily yield the tightest bounding box, but it is general and straightforward to implement.

10.9 Sub-Linear Ray-Object Intersection

In the earlier ray-object intersection pseudocode, all objects are looped over, checking for intersections. For N objects, this is an $O(N)$ linear search and is thus slow for large values of N . 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 partition-

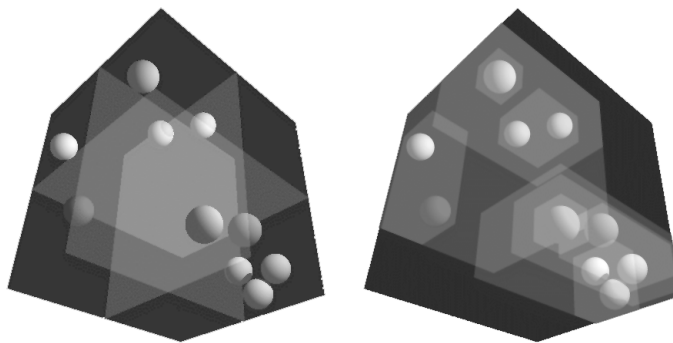


Figure 10.14. Left: a uniform partitioning of space. Right: adaptive bounding-box hierarchy. *Image courtesy David DeMarle.*

ing (Glassner, 1984; Jansen, 1986; Havran, 2000). An example of the first two strategies is shown in Figure 10.14.

10.9.1 Bounding Boxes

A key operation in most intersection acceleration schemes is computing the intersection of a ray with a bounding box (Figure 10.15). 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.

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:

$$x = x_{\min},$$

$$x = x_{\max},$$

$$y = y_{\min},$$

$$y = y_{\max}.$$

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

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

As shown in Figure 10.16, the intersection test can be phrased in terms of these intervals. First, we compute the ray parameter where the ray hits the line $x = x_{\min}$:

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

We then make similar computations for $t_{x\max}$, $t_{y\min}$, and $t_{y\max}$. The ray hits the box if and only if the intervals $[t_{x\min}, t_{x\max}]$ and $[t_{y\min}, t_{y\max}]$ overlap, i.e., their intersection is non-empty. In pseudocode this algorithm is:

```

 $t_{x\min} = (x_{\min} - x_e) / x_d$ 
 $t_{x\max} = (x_{\max} - x_e) / x_d$ 
 $t_{y\min} = (y_{\min} - y_e) / y_d$ 
 $t_{y\max} = (y_{\max} - y_e) / y_d$ 
if ( $t_{x\min} > t_{y\max}$ ) or ( $t_{y\min} > t_{x\max}$ ) then
  return false
else
  return true

```

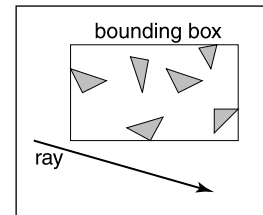


Figure 10.15. The ray is only tested for intersection with the surfaces if it hits the bounding box.

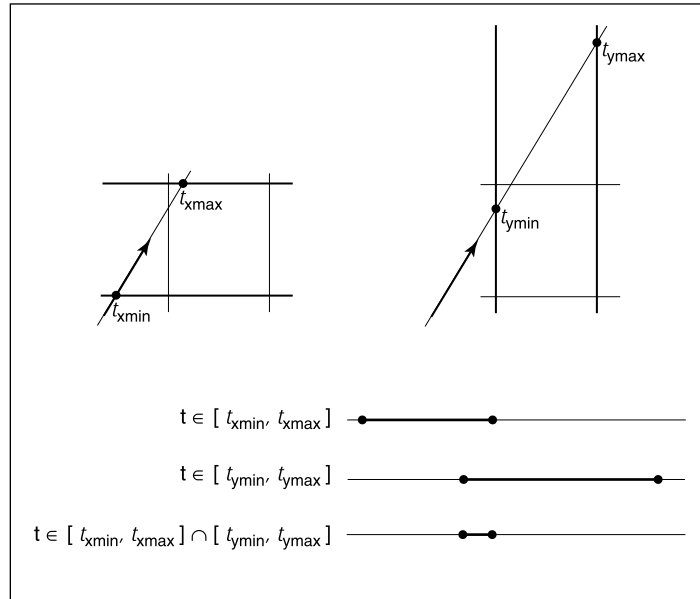


Figure 10.16. The ray will be inside the interval $x \in [x_{min}, x_{max}]$ for some interval in its parameter space $t \in [t_{xmin}, t_{xmax}]$. A similar interval exists for the y interval. The ray intersects the box if it is in both the x interval and y interval at the same time, i.e., the intersection of the two one-dimensional intervals is not empty.

The 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.

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 1.6 the rules for divide by zero: for any positive real number a ,

$$\begin{aligned} +a/0 &= +\infty; \\ -a/0 &= -\infty. \end{aligned}$$

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

$$\begin{aligned} t_{x\min} &= \frac{x_{\min} - x_e}{0}; \\ t_{x\max} &= \frac{x_{\max} - x_e}{0}. \end{aligned}$$

There are three possibilities of interest:

1. $x_e \leq x_{\min}$ (no hit);
2. $x_{\min} < x_e < x_{\max}$ (hit);
3. $x_{\max} \leq x_e$ (no hit).

For the first case we have

$$\begin{aligned} t_{x\min} &= \frac{\text{positive number}}{0}; \\ t_{x\max} &= \frac{\text{positive number}}{0}. \end{aligned}$$

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

$$\begin{aligned} t_{x\min} &= \frac{\text{negative number}}{0}; \\ t_{x\max} &= \frac{\text{positive number}}{0}. \end{aligned}$$

This yields the interval $(t_{x\min}, t_{x\min}) = (-\infty, \infty)$ 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
     $t_{\min} = (x_{\min} - x_e) / x_d$ 
     $t_{\max} = (x_{\max} - x_e) / x_d$ 
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
     $t_{\min} = a(x_{\max} - x_e)$ 
     $t_{\max} = a(x_{\min} - x_e)$ 

```

10.9.2 Hierarchical Bounding Boxes

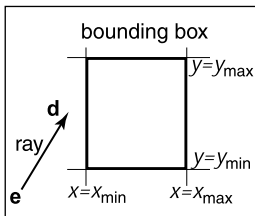


Figure 10.17. A 2D ray $e + td$ is tested against a 2D bounding box.

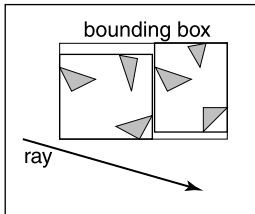


Figure 10.18. The bounding boxes can be nested by creating boxes around subsets of the model.

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 10.17. 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 10.18. The data structure for the hierarchy shown in Figure 10.19 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 10.19. 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.

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:

```

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

```

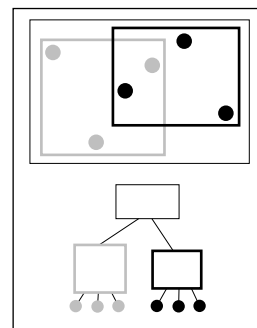


Figure 10.19. The grey box is a tree node that points to the three grey spheres, and the thick black box points to the three black spheres. Note that not all spheres enclosed by the box are guaranteed to be pointed to by the corresponding tree node.

```

    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:

```

bvh-node::bvh-node(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-node → bbox, right-node → 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:

```

bvh-node::bvh-node(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 node(A[0..k], (AXIS +1) mod 3)
  right = new node(A[k+1..N-1], (AXIS +1) mod 3)
  bbox = combine(left-node → bbox, right-node → 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.

10.9.3 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:

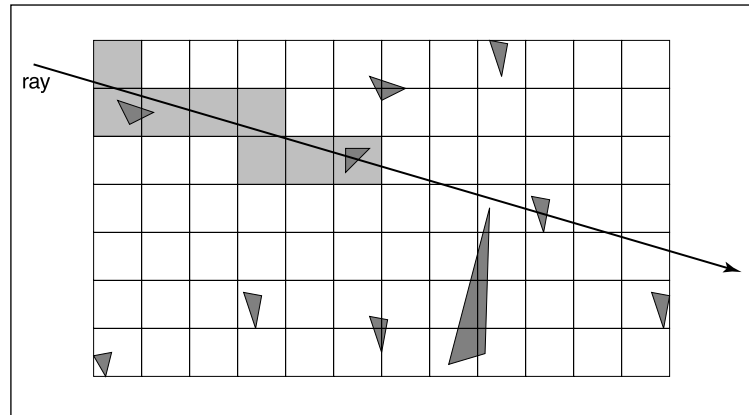


Figure 10.20. In uniform spatial subdivision, the ray is tracked forward through cells until an object in one of those cells is hit. In this example, only objects in the shaded cells are checked.

- 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.

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 10.20. When an object is hit, the traversal ends.

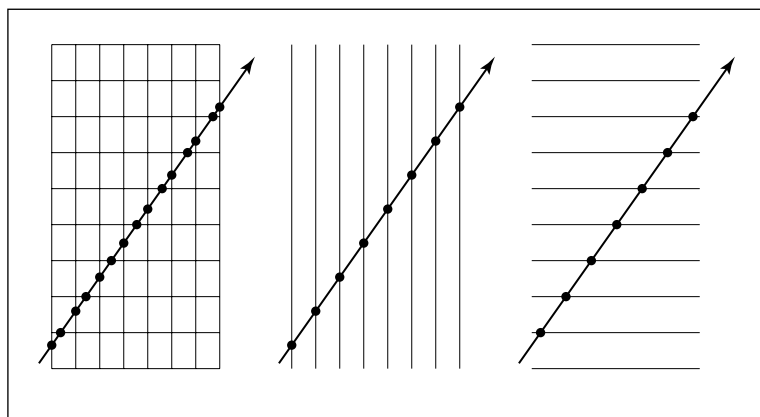


Figure 10.21. Although the pattern of cell hits seems irregular (left), the hits on sets of parallel planes are very even.

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 10.21. 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_{\min}, y_{\min}) and (x_{\max}, y_{\max}) . The grid has n_x by n_y cells.

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 10.22). 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 10.23). 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 13.4.

10.9.4 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 a painter’s algorithm in Chapter 8, but it usually uses axis-aligned cutting planes for easier ray intersection. A node in this structure might contain a single cutting plane and a left and right subtree. These subtrees would contain all objects on either side of the cutting plane. Objects that pass through the plane would be in each subtree. 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
```

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 10.24. For our purposes, the origin of these rays is a point at parameter t_0 :

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

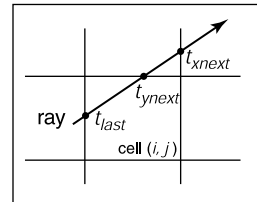


Figure 10.22. To decide whether we advance right or upwards, we keep track of the intersections with the next vertical and horizontal boundary of the cell.

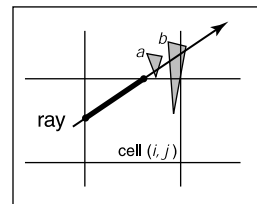


Figure 10.23. Only hits within the cell should be reported. Otherwise the case above would cause us to report hitting object b rather than object a .

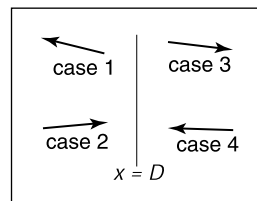


Figure 10.24. The four cases of how a ray relates to the BSP cutting plane $x = D$.

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:

```

bool bsp-node::hit( ray a + tb, real t0, real t1, hit-record rec)
xp = xa + t0xb
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.

10.10 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 10.25. 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 10.26, 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.

10.11 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

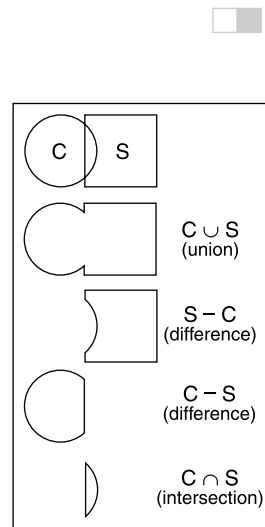


Figure 10.25. The basic CSG operations on a 2D circle and square.

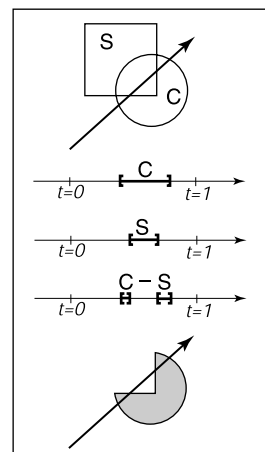


Figure 10.26. Intervals are processed to indicate how the ray hits the composite object.

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 3.7) the image.

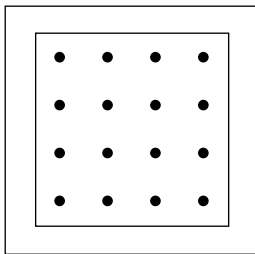


Figure 10.27. Sixteen regular samples for a single pixel.

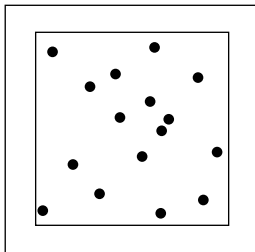


Figure 10.28. Sixteen random samples for a single pixel.

10.11.1 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
   $c_{ij} = \text{ray-color}(i + 0.5, j + 0.5)$ 
```

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
       $c = c + \text{ray-color}(i + (p + 0.5)/n, j + (q + 0.5)/n)$ 
   $c_{ij} = c/n^2$ 
```

This is usually called *regular sampling*. The 16 sample locations in a pixel for $n = 4$ are shown in Figure 10.27. 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 10.28. This is usually called *random sampling* and involves just a small change to the code:

```

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

```

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
       $c = c + \text{ray-color}(i + (p + \xi)/n, j + (q + \xi)/n)$ 
   $c_{ij} = c/n^2$ 

```

That method is usually called *jittering* or *stratified sampling* (Figure 10.29).

10.11.2 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 non-zero area and can thus be partially visible. This idea is shown in 2D in Figure 10.30. The region where the light is entirely invisible is called the *umbra*. The partially visible region is called the *penumbra*. There is not a commonly used term for the region not in shadow, but it is sometimes called the *anti-umbra*.

The key to implementing soft shadows is to somehow account for the light being an area rather than a point. An easy way to do this is to approximate the light with a distributed set of N point lights each with one N th of the intensity of the base light. This concept is illustrated at the left of Figure 10.31 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.

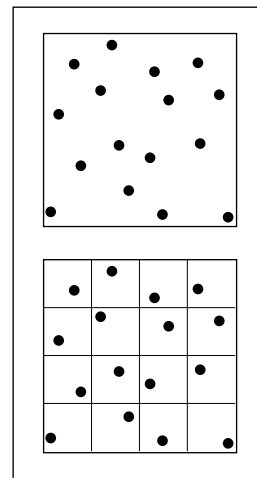


Figure 10.29. Sixteen stratified (jittered) samples for a single pixel shown with and without the bins highlighted. There is exactly one random sample taken within each bin.

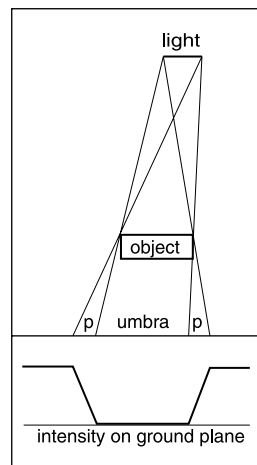


Figure 10.30. A soft shadow has a gradual transition from the unshadowed to shadowed region. The transition zone is the “penumbra” denoted by p in the figure.

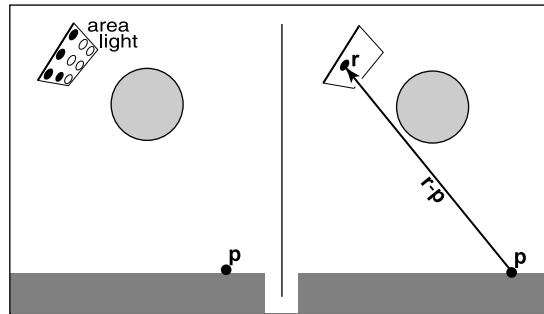


Figure 10.31. Left: an area light can be approximated by some number of point lights; four of the nine points are visible to p so it is in the penumbra. Right: a random point on the light is chosen for the shadow ray, and it has some chance of hitting the light or not.

This amounts to choosing a random point on the light for any surface point being lit as is shown at the right of Figure 10.31.

If the light is a parallelogram specified by a corner point c and two edge vectors a and b (Figure 10.32), then choosing a random point r is straightforward:

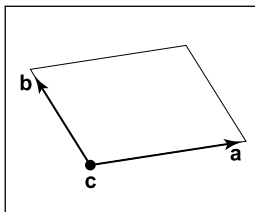


Figure 10.32. The geometry of a parallelogram light specified by a corner point and two edge vectors.

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

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

```

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

```


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  $j$ 

```

10.11.3 Depth of Field

The soft focus effects seen in most photos can be simulated by collecting light at a non-zero 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 10.33). We can place the “window” we are sampling on the plane where everything is in focus (rather than at the $z = n$ plane as we did previously), and the lens at the eye. The distance to the plane where everything is in focus we call the *focus plane*, and the distance to it is set by the user, just as the distance to the focus plane in a real camera is set by the user or range finder.

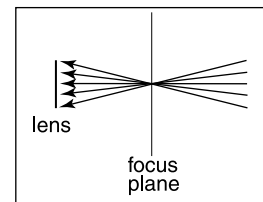


Figure 10.33. The lens averages over a cone of directions that hit the pixel location being sampled.



Figure 10.34. 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. (See also Plate VII.)

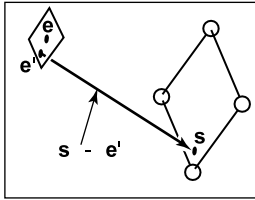


Figure 10.35. To create depth-of-field effects, the eye is randomly selected from a square region.

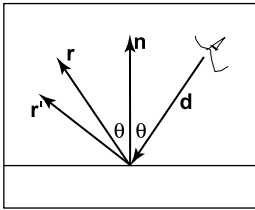


Figure 10.36. The reflection ray is perturbed to a random vector \mathbf{r}' .

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 10.35). 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 10.34.

10.11.4 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 10.36.

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 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.

10.11.5 Motion Blur

We can add a blurred appearance to objects as shown in Figure 10.37. This is called *motion blur* and is the result of the image being formed over a non-zero

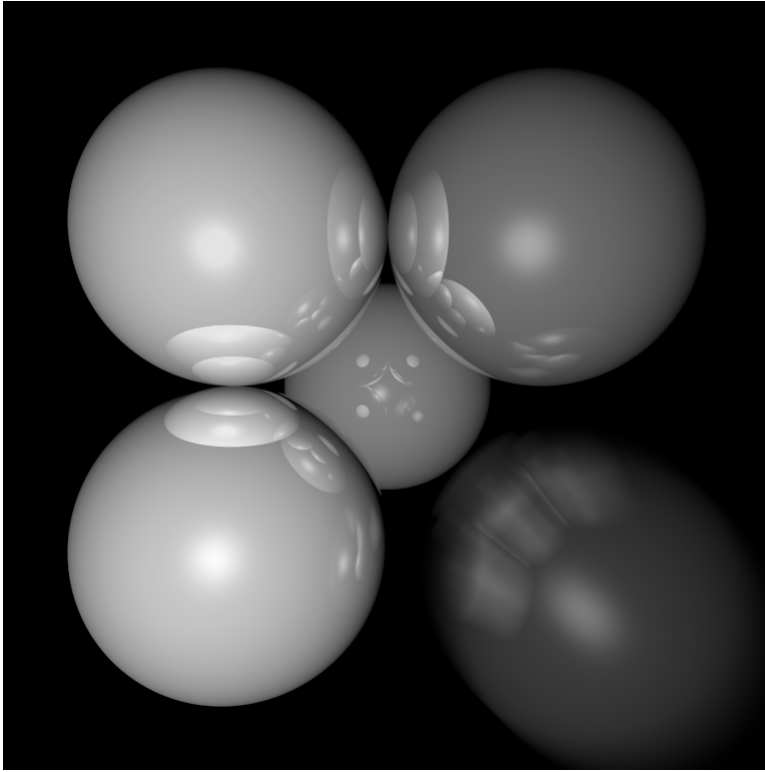


Figure 10.37. The bottom right sphere is in motion, and a blurred appearance results. *Image courtesy Chad Barb.*

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).

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.

- 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. 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.

- 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.

