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.

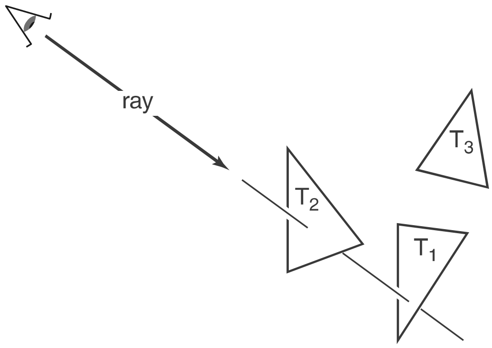

Figure 4.1. The ray is "traced" into the scene and the first object hit is the one seen through the pixel. In this case, the triangle T_2 is returned.

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 $\mathbf{n}$
set pixel color to value computed from hit point, light, and $\mathbf{n}$

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 fish-eye 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 Greenspan.*

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

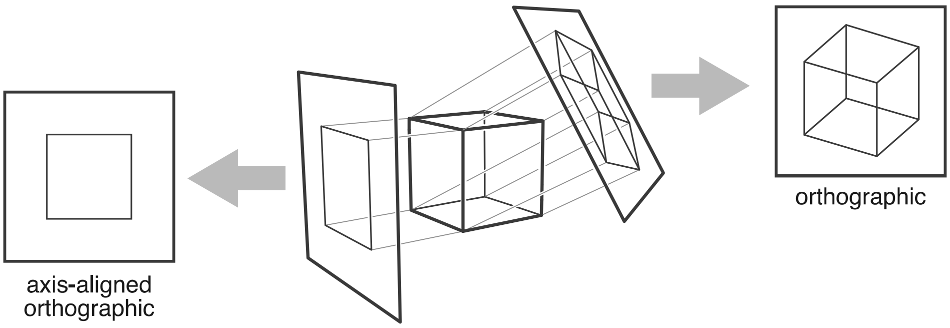

Figure 4.3. When projection lines are parallel and perpendicular to the image plane, the resulting views are called orthographic.

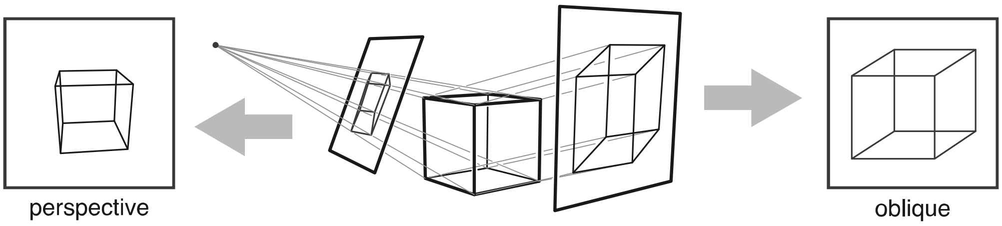

Figure 4.4. A parallel projection that has the image plane at an angle to the projection direction is called oblique (right). In perspective projection, the projection lines all pass through the viewpoint, rather than being parallel (left). The illustrated perspective view is non-oblique because a projection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

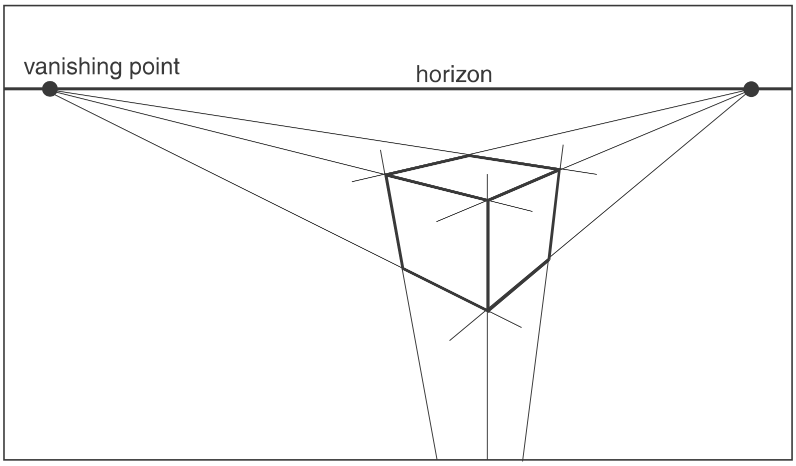

Figure 4.5. In three-point perspective, an artist picks "vanishing points" where parallel lines meet. Parallel horizontal lines will meet at a point on the horizon. Every set of parallel lines has its own vanishing points. These rules are followed automatically if we implement perspective based on the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

looking views using *perspective projection*: we simply project along lines that pass through a single point, the *viewpoint*, rather than along parallel lines (Figure 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

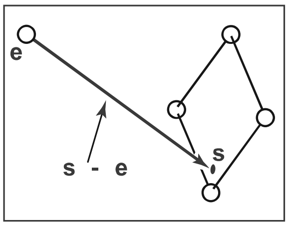

Figure 4.6. The ray from the eye to a point on the image plane.

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

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 $\mathbf{p}$. The point $\mathbf{e}$ is the ray's *origin*, and $\mathbf{s} - \mathbf{e}$ is the ray's *direction*.

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

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.

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.

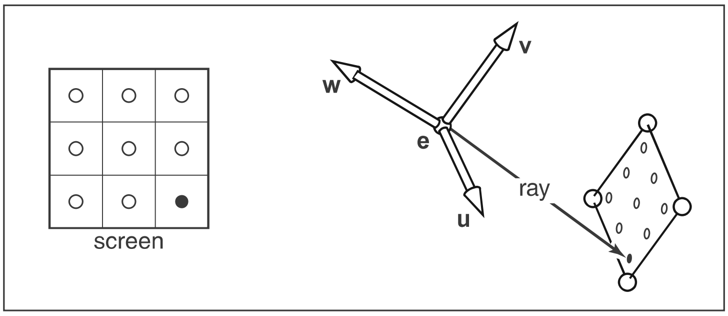

Figure 4.7. The sample points on the screen are mapped to a similar array on the 3D window. A viewing ray is sent to each of these locations.

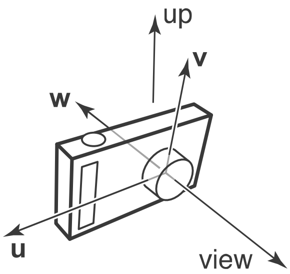

Figure 4.8. The vectors of the camera frame, together with the view direction and up direction. The **w** vector is opposite the view direction, and the **v** vector is coplanar with **w** and the up vector.

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.

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 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,

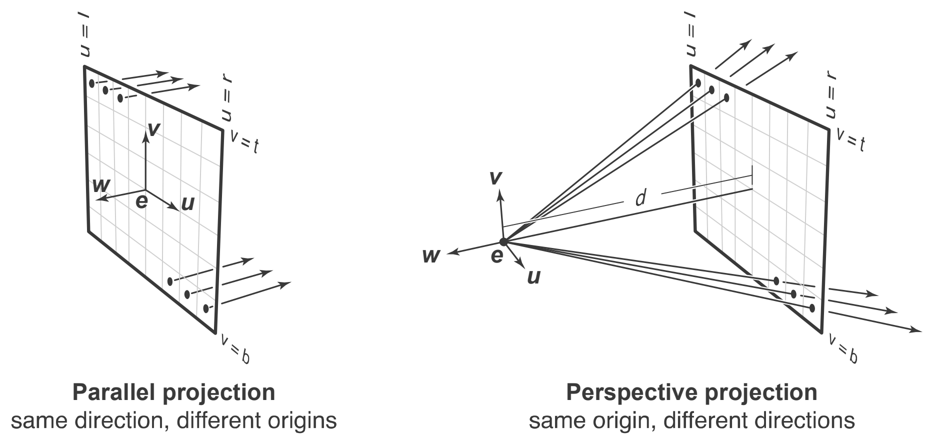

Figure 4.9. Ray generation using the camera frame. Left: In an orthographic view, the rays start at the pixels' locations on the image plane, and all share the same direction, which is equal to the view direction. Right: In a perspective view, the rays start at the viewpoint, and each ray's direction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

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

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

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 **v**-axis).

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 $\leftarrow \mathbf{e} + u\,\mathbf{u} + v\,\mathbf{v}$

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:

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

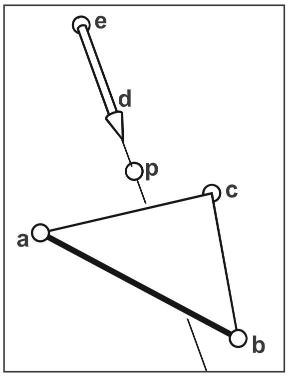

Figure 4.10. The ray hits the plane containing the triangle at point **p**.

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

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

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, 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; 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 **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 $t \in [t_0, t_1]$) **then**
    hit = true
    hitobject = **o**
    $t_1 = t$
return hit

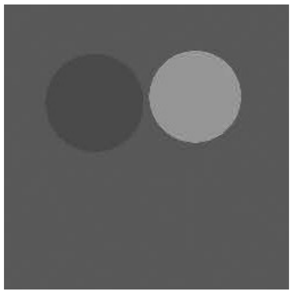

Figure 4.11. A simple scene rendered with only ray generation and surface intersection, but no shading; each pixel is just set to a fixed color depending on which object it hit.

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 towards 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 towards 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*:

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

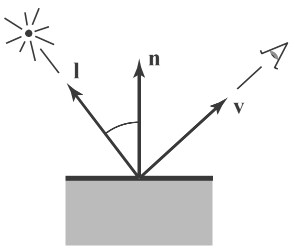

Figure 4.12. Geometry for Lambertian shading.

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

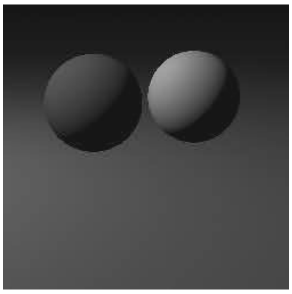

Figure 4.13. A simple scene rendered with diffuse shading from a single light source.

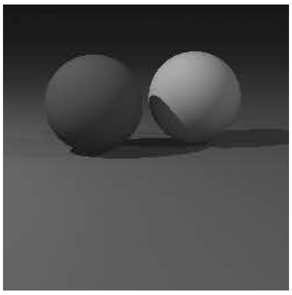

Figure 4.14. A simple scene rendered with diffuse shading and shadows (Section 4.7) from three light sources.

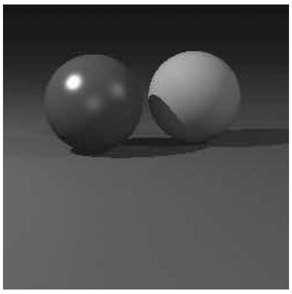

Figure 4.15. A simple scene rendered with diffuse shading (right), Blinn-Phong shading (left), and shadows (Section 4.7) from three light sources.

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 refelction then decreases smoothly as the vectors move away from a mirror configuration.

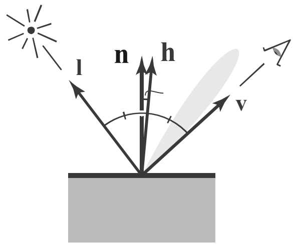

Figure 4.16. Geometry for Blinn-Phong shading.

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 $\mathbf{h}$.

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

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

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

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

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

4.5.3 Ambient Shading

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

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 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 i^{th} 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 $\mathbf{e} + t\mathbf{d}$, real t_0, real t_1, 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

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.

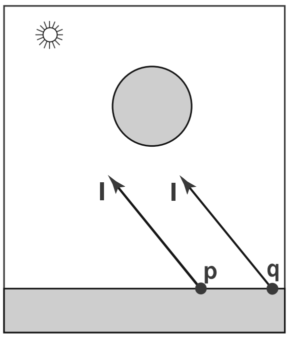

Figure 4.17. The point **p** is not in shadow while the point **q** is in shadow.

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).

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

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 $c = \text{rec}.k_a\ I_a$
  if (not scene→hit($\mathbf{p} + s\mathbf{l}$, ϵ, ∞, srec)) **then**
    vector3 $\mathbf{h} = \text{normalized}(\text{normalized}(\mathbf{l}) + \text{normalized}(-\mathbf{d}))$
    $c = c + \text{rec}.k_d\, I \max(0, \text{rec}.\mathbf{n} \cdot \mathbf{l}) + (\text{rec}.k_s)\, I\, (\text{rec}.\mathbf{n} \cdot \mathbf{h})^{\text{rec}.p}$
  return c
else
  return background-color

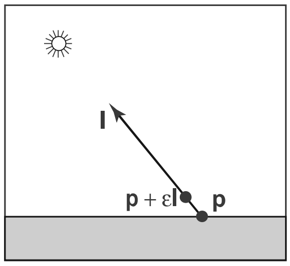

Figure 4.18. By testing in the interval starting at ϵ, we avoid numerical imprecision causing the ray to hit the surface **p** is on.

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

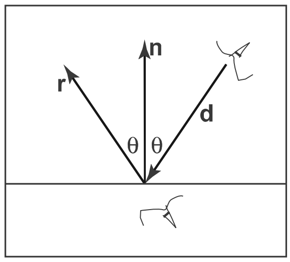

Figure 4.19. When looking into a perfect mirror, the viewer looking in direction **d** will see whatever the viewer "below" the surface would see in direction **r**.

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,

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

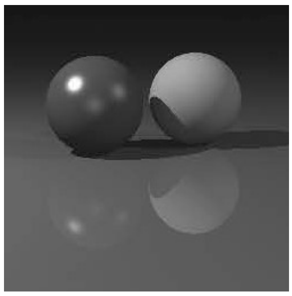

Figure 4.20. A simple scene rendered with diffuse and Blinn-Phong shading, shadows from three light sources, and specular reflection from the floor.

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

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).

4.9 Historical Notes

Ray tracing was developed early in the history of computer graphics (Appel, 1968) but was not used much until a while later when 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; Parker, Martin, et al., 1999; Wald et al., 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.