Implicit Quadric Curves

In the previous section we saw that a linear function $f(x, y)$ gives rise to an implicit line $f(x, y) = 0$. If f is instead a quadratic function of x and y, with the general form

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0,$$

the resulting implicit curve is called a quadric. Two-dimensional quadric curves include ellipses and hyperbolas, as well as the special cases of parabolas, circles, and lines.

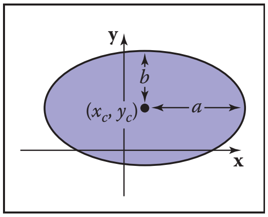


Figure 2.32. The ellipse with center (x_c, y_c) and semi-axes of length a and b.

Examples of quadric curves include the circle with center (x_c, y_c) and radius r,

$$(x - x_c)^2 + (y - y_c)^2 - r^2 = 0,$$

and axis-aligned ellipses of the form

Try setting $a = b = r$ in the ellipse equation and compare to the circle equation.

$$\frac{(x - x_c)^2}{a^2} + \frac{(y - y_c)^2}{b^2} - 1 = 0,$$

where (x_c, y_c) is the center of the ellipse, and a and b are the minor and major semi-axes (Figure 2.32).

2.5.3 3D Implicit Surfaces

Just as implicit equations can be used to define curves in 2D, they can be used to define surfaces in 3D. As in 2D, implicit equations *implicitly* define a set of points that are on the surface:

$$f(x, y, z) = 0.$$

Any point (x, y, z) that is on the surface results in zero when given as an argument to f. Any point not on the surface results in some number other than zero. You can check whether a point is on the surface by evaluating f, or you can check which side of the surface the point lies on by looking at the sign of f, but you cannot always explicitly construct points on the surface. Using vector notation, we will write such functions of $\mathbf{p} = (x, y, z)$ as

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

2.5.4 Surface Normal to an Implicit Surface

A surface normal (which is needed for lighting computations, among other things) is a vector perpendicular to the surface. Each point on the surface may have a different normal vector. In the same way that the gradient provides a normal to an implicit curve in 2D, the surface normal at a point $\mathbf{p}$ on an implicit surface is given by the gradient of the implicit function

$$\mathbf{n} = \nabla f(\mathbf{p}) = \left(\frac{\partial f(\mathbf{p})}{\partial x}, \frac{\partial f(\mathbf{p})}{\partial y}, \frac{\partial f(\mathbf{p})}{\partial z} \right).$$

The reasoning is the same as for the 2D case: the gradient points in the direction of fastest increase in f, which is perpendicular to all directions tangent to the surface, in which f remains constant. The gradient vector points toward the side of the surface where $f(\mathbf{p}) > 0$, which we may think of as "into" the surface or "out from" the surface in a given context. If the particular form of f creates inward-facing gradients, and outward-facing gradients are desired, the surface $-f(\mathbf{p}) = 0$ is the same as surface $f(\mathbf{p}) = 0$ but has directionally reversed gradients, i.e., $-\nabla f(\mathbf{p}) = \nabla(-f(\mathbf{p}))$.

2.5.5 Implicit Planes

As an example, consider the infinite plane through point $\mathbf{a}$ with surface normal $\mathbf{n}$. The implicit equation to describe this plane is given by

$$(\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0. \tag{2.21}$$

Note that $\mathbf{a}$ and $\mathbf{n}$ are known quantities. The point $\mathbf{p}$ is any unknown point that satisfies the equation. In geometric terms this equation says "the vector from $\mathbf{a}$ to $\mathbf{p}$ is perpendicular to the plane normal." If $\mathbf{p}$ were not in the plane, then $(\mathbf{p} - \mathbf{a})$ would not make a right angle with $\mathbf{n}$ (Figure 2.33).

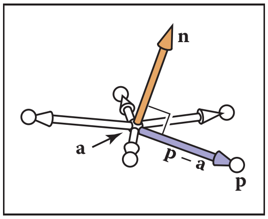


Figure 2.33. Any of the points $\mathbf{p}$ shown are in the plane with normal vector $\mathbf{n}$ that includes point $\mathbf{a}$ if Equation (2.2) is satisfied.

Sometimes we want the implicit equation for a plane through points $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$. The normal to this plane can be found by taking the cross product of any two vectors in the plane. One such cross product is

$$\mathbf{n} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a}).$$

This allows us to write the implicit plane equation:

$$(\mathbf{p} - \mathbf{a}) \cdot ((\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})) = 0. \tag{2.22}$$

A geometric way to read this equation is that the volume of the parallelepiped defined by $\mathbf{p} - \mathbf{a}$, $\mathbf{b} - \mathbf{a}$, and $\mathbf{c} - \mathbf{a}$ is zero, i.e., they are coplanar. This can only be true if $\mathbf{p}$ is in the same plane as $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$. The full-blown Cartesian representation for this is given by the determinant (this is discussed in more detail in Section 5.3):

$$\begin{vmatrix} x - x_a & y - y_a & z - z_a \\ x_b - x_a & y_b - y_a & z_b - z_a \\ x_c - x_a & y_c - y_a & z_c - z_a \end{vmatrix} = 0. \tag{2.23}$$

The determinant can be expanded (see Section 5.3 for the mechanics of expanding determinants) to the bloated form with many terms.

Equations (2.22) and (2.23) are equivalent, and comparing them is instructive. Equation (2.22) is easy to interpret geometrically and will yield efficient code. In addition, it is relatively easy to avoid a typographic error that compiles into incorrect code if it takes advantage of debugged cross and dot product code. Equation (2.23) is also easy to interpret geometrically and will be efficient provided an efficient 3 × 3 determinant function is implemented. It is also easy to implement without a typo if a function *determinant*$(\mathbf{a}, \mathbf{b}, \mathbf{c})$ is available. It will be especially easy for others to read your code if you rename the *determinant* function *volume*. So both Equations (2.22) and (2.23) map well into code. The full expansion of either equation into x-, y-, and z-components is likely to generate typos. Such typos are likely to compile and, thus, to be especially pesky. This is an excellent example of clean math generating clean code and bloated math generating bloated code.

3D Quadric Surfaces

Just as quadratic polynomials in two variables define quadric curves in 2D, quadratic polynomials in x, y, and z define *quadric surfaces* in 3D. For instance, a sphere can be written as

$$f(\mathbf{p}) = (\mathbf{p} - \mathbf{c})^2 - r^2 = 0,$$

and an axis-aligned ellipsoid may be written as

$$f(\mathbf{p}) = \frac{(x - x_c)^2}{a^2} + \frac{(y - y_c)^2}{b^2} + \frac{(z - z_c)^2}{c^2} - 1 = 0.$$

3D Curves from Implicit Surfaces

One might hope that an implicit 3D curve could be created with the form $f(\mathbf{p}) = 0$. However, all such curves are just degenerate surfaces and are rarely useful in practice. A 3D curve can be constructed from the intersection of two simultaneous implicit equations:

$$\begin{aligned} f(\mathbf{p}) &= 0, \\ g(\mathbf{p}) &= 0. \end{aligned}$$

For example, a 3D line can be formed from the intersection of two implicit planes. Typically, it is more convenient to use parametric curves instead; they are discussed in the following sections.

2.5.6 2D Parametric Curves

A *parametric* curve is controlled by a single *parameter* that can be considered a sort of index that moves continuously along the curve. Such curves have the form

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} g(t) \\ h(t) \end{bmatrix}.$$

Here (x, y) is a point on the curve, and t is the parameter that influences the curve. For a given t, there will be some point determined by the functions g and h. For continuous g and h, a small change in t will yield a small change in x and y. Thus, as t continuously changes, points are swept out in a continuous curve. This is a nice feature because we can use the parameter t to explicitly construct points on the curve. Often we can write a parametric curve in vector form,

$$\mathbf{p} = f(t),$$

where f is a vector-valued function, $f : \mathbb{R} \mapsto \mathbb{R}^2$. Such vector functions can generate very clean code, so they should be used when possible.

We can think of the curve with a position as a function of time. The curve can go anywhere and could loop and cross itself. We can also think of the curve as having a velocity at any point. For example, the point $\mathbf{p}(t)$ is traveling slowly near $t = -2$ and quickly between $t = 2$ and $t = 3$. This type of “moving point” vocabulary is often used when discussing parametric curves even when the curve is not describing a moving point.

2D Parametric Lines

A parametric line in 2D that passes through points $\mathbf{p}_0 = (x_0, y_0)$ and $\mathbf{p}_1 = (x_1, y_1)$ can be written as

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_0 + t(x_1 - x_0) \\ y_0 + t(y_1 - y_0) \end{bmatrix}.$$

Because the formulas for x and y have such similar structure, we can use the vector form for $\mathbf{p} = (x, y)$ (Figure 2.34):

$$\mathbf{p}(t) = \mathbf{p}_0 + t(\mathbf{p}_1 - \mathbf{p}_0).$$

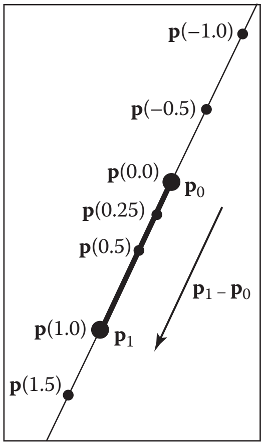


Figure 2.34. A 2D parametric line through $\mathbf{p}_0$ and $\mathbf{p}_1$. The line segment defined by $t \in [0,1]$ is shown in bold.

You can read this in geometric form as: "start at point $\mathbf{p}_0$ and go some distance toward $\mathbf{p}_1$ determined by the parameter t." A nice feature of this form is that $\mathbf{p}(0) = \mathbf{p}_0$ and $\mathbf{p}(1) = \mathbf{p}_1$. Since the point changes linearly with t, the value of t between $\mathbf{p}_0$ and $\mathbf{p}_1$ measures the fractional distance between the points. Points with $t < 0$ are to the "far" side of $\mathbf{p}_0$, and points with $t > 1$ are to the "far" side of $\mathbf{p}_1$.

Parametric lines can also be described as just a point $\mathbf{o}$ and a vector $\mathbf{d}$:

$$\mathbf{p}(t) = \mathbf{o} + t(\mathbf{d}).$$

When the vector $\mathbf{d}$ has unit length, the line is *arc-length parameterized*. This means t is an exact measure of distance along the line. Any parametric curve can be arc-length parameterized, which is obviously a very convenient form, but not all can be converted analytically.

2D Parametric Circles

A circle with center (x_c, y_c) and radius r has a parametric form:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_c + r\cos\phi \\ y_c + r\sin\phi \end{bmatrix}.$$

To ensure that there is a unique parameter ϕ for every point on the curve, we can restrict its domain: $\phi \in [0, 2\pi)$ or $\phi \in (-\pi, \pi]$ or any other half-open interval of length 2π.

An axis-aligned ellipse can be constructed by scaling the x and y parametric equations separately:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_c + a\cos\phi \\ y_c + b\sin\phi \end{bmatrix}.$$

2.5.7 3D Parametric Curves

A 3D parametric curve operates much like a 2D parametric curve:

$$\begin{aligned} x &= f(t), \\ y &= g(t), \\ z &= h(t). \end{aligned}$$

For example, a spiral around the z-axis is written as:

$$\begin{aligned} x &= \cos t, \\ y &= \sin t, \\ z &= t. \end{aligned}$$

As with 2D curves, the functions f, g, and h are defined on a domain $D \subset \mathbb{R}$ if we want to control where the curve starts and ends. In vector form we can write

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \mathbf{p}(t).$$

In this chapter we only discuss 3D parametric lines in detail. General 3D parametric curves are discussed more extensively in Chapter 15.

The parametric curve is the range of $\mathbf{p}$: $\mathbb{R} \to \mathbb{R}^3$.

3D Parametric Lines

A 3D parametric line can be written as a straightforward extension of the 2D parametric line, e.g.,

$$\begin{aligned} x &= 2 + 7t, \\ y &= 1 + 2t, \\ z &= 3 - 5t. \end{aligned}$$

This is cumbersome and does not translate well to code variables, so we will write it in vector form:

$$\mathbf{p} = \mathbf{o} + t\mathbf{d},$$

where, for this example, $\mathbf{o}$ and $\mathbf{d}$ are given by

$$\begin{aligned} \mathbf{o} &= (2, 1, \quad 3), \\ \mathbf{d} &= (7, 2, -5). \end{aligned}$$

Note that this is very similar to the 2D case. The way to visualize this is to imagine that the line passes through $\mathbf{o}$ and is parallel to $\mathbf{d}$. Given any value of t, you get some point $\mathbf{p}(t)$ on the line. For example, at $t = 2$, $p(t) = (2, 1, 3) + 2(7, 2, -5) = (16, 5, -7)$. This general concept is the same as for two dimensions (Figure 2.30).

As in 2D, a *line segment* can be described by a 3D parametric line and an interval $t \in [t_a, t_b]$. The line segment between two points $\mathbf{a}$ and $\mathbf{b}$ is given by $\mathbf{p}(t) = \mathbf{a} + t(\mathbf{b} - \mathbf{a})$ with $t \in [0, 1]$. Here $\mathbf{p}(0) = \mathbf{a}$, $\mathbf{p}(1) = \mathbf{b}$, and $\mathbf{p}(0.5) = (\mathbf{a} + \mathbf{b})/2$, the midpoint between $\mathbf{a}$ and $\mathbf{b}$.

A *ray*, or *half-line*, is a 3D parametric line with a half-open interval, usually $[0, \infty)$. From now on we will refer to all lines, line segments, and rays as “rays.” This is sloppy, but corresponds to common usage and makes the discussion simpler.

2.5.8 3D Parametric Surfaces

The parametric approach can be used to define surfaces in 3D space in much the same way we define curves, except that there are two parameters to address the two-dimensional area of the surface. These surfaces have the form

$$\begin{aligned} x &= f(u, v), \\ y &= g(u, v), \\ z &= h(u, v). \end{aligned}$$

The parametric surface is the range of the function $\mathbf{p}$: $\mathbb{R}^2 \to \mathbb{R}^3$.

or, in vector form,

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \mathbf{p}(u, v).$$

Pretend for the sake of argument that the Earth is exactly spherical.

The θ and ϕ here may or may not seem reversed depending on your background; the use of these symbols varies across disciplines. In this book we will always assume the meaning of θ and ϕ used in Equation (2.24) and depicted in Figure 2.35.

Example. For example, a point on the surface of the Earth can be described by the two parameters longitude and latitude. If we define the origin to be at the center of the Earth, and let r be the radius of the Earth, then a spherical coordinate system centered at the origin (Figure 2.35), lets us derive the parametric equations

$$\begin{aligned} x &= r \cos\phi \sin\theta, \\ y &= r \sin\phi \sin\theta, \\ z &= r \cos\theta. \end{aligned} \tag{2.24}$$

Ideally, we'd like to write this in vector form, but it isn't feasible for this particular parametric form.

We would also like to be able to find the (θ, ϕ) for a given (x, y, z). If we assume that $\phi \in (-\pi, \pi]$ this is easy to do using the *atan2* function from Equation (2.2):

$$\begin{aligned} \theta &= \text{acos}(z/\sqrt{x^2 + y^2 + z^2}), \\ \phi &= \text{atan2}(y, x). \end{aligned} \tag{2.25}$$

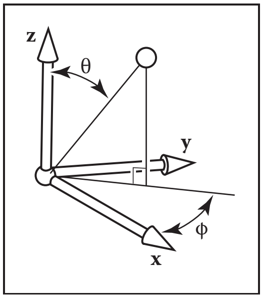


Figure 2.35. The geometry for spherical coordinates.

With implicit surfaces, the derivative of the function f gave us the surface normal. With parametric surfaces, the derivatives of $\mathbf{p}$ also give information about the surface geometry.

Consider the function $\mathbf{q}(t) = \mathbf{p}(t, v_0)$. This function defines a parametric curve obtained by varying u while holding v fixed at the value v_0. This curve, called an *isoparametric curve* (or sometimes "isoparm" for short) lies in the surface. The derivative of $\mathbf{q}$ gives a vector tangent to the curve, and since the curve lies in the surface the vector $\mathbf{q}'$ also lies in the surface. Since it was obtained by varying one argument of $\mathbf{p}$, the vector $\mathbf{q}'$ is the partial derivative of $\mathbf{p}$ with respect to u, which we'll denote $\mathbf{p}_u$. A similar argument shows that the partial derivative $\mathbf{p}_v$ gives the tangent to the isoparametric curves for constant u, which is a second tangent vector to the surface.

The derivative of $\mathbf{p}$, then, gives two tangent vectors at any point on the surface. The normal to the surface may be found by taking the cross product of these vectors: since both are tangent to the surface, their cross product, which is perpendicular to both tangents, is normal to the surface. The right-hand rule for cross products provides a way to decide which side is the front, or outside, of the surface; we will use the convention that the vector

$$\mathbf{n} = \mathbf{p}_u \times \mathbf{p}_v$$

points toward the outside of the surface.

2.5.9 Summary of Curves and Surfaces

Implicit curves in 2D or surfaces in 3D are defined by scalar-valued functions of two or three variables, $f : \mathbb{R}^2 \to \mathbb{R}$ or $f : \mathbb{R}^3 \to \mathbb{R}$, and the surface consists of all points where the function is zero:

$$S = \{\mathbf{p} \,|\, f(\mathbf{p}) = 0\}.$$

Parametric curves in 2D or 3D are defined by vector-valued functions of one variable, $\mathbf{p} : D \subset \mathbb{R} \to \mathbb{R}^2$ or $\mathbf{p} : D \subset \mathbb{R} \to \mathbb{R}^3$, and the curve is swept out as t varies over all of D:

$$S = \{\mathbf{p}(t) \,|\, t \in D\}.$$

Parametric surfaces in 3D are defined by vector-valued functions of two variables, $\mathbf{p} : D \subset \mathbb{R}^2 \to \mathbb{R}^3$, and the surface consists of the images of all points (u, v) in the domain:

$$S = \{\mathbf{p}(t) \,|\, (u, v) \in D\}.$$

For implicit curves and surfaces, the normal vector is given by the derivative of f (the gradient), and the tangent vector (for a curve) or vectors (for a surface) can be derived from the normal by constructing a basis.

For parametric curves and surfaces, the derivative of $\mathbf{p}$ gives the tangent vector (for a curve) or vectors (for a surface), and the normal vector can be derived from the tangents by constructing a basis.

2.6 Linear Interpolation

Perhaps the most common mathematical operation in graphics is *linear interpolation*. We have already seen an example of linear interpolation of position to form line segments in 2D and 3D, where two points $\mathbf{a}$ and $\mathbf{b}$ are associated with a parameter t to form the line $\mathbf{p} = (1-t)\mathbf{a} + t\mathbf{b}$. This is *interpolation* because $\mathbf{p}$ goes through $\mathbf{a}$ and $\mathbf{b}$ exactly at $t = 0$ and $t = 1$. It is *linear* interpolation because the weighting terms t and $1 - t$ are linear polynomials of t.

Another common linear interpolation is among a set of positions on the x-axis: $x_0, x_1, \ldots, x_n$, and for each x_i we have an associated height, y_i. We want to create a continuous function $y = f(x)$ that interpolates these positions, so that f goes through every data point, i.e., $f(x_i) = y_i$. For linear interpolation, the points (x_i, y_i) are connected by straight line segments. It is natural to use parametric line equations for these segments. The parameter t is just the fractional distance between x_i and x_{i+1}:

$$f(x) = y_i + \frac{x - x_i}{x_{i+1} - x_i}(y_{i+1} - y_i). \tag{2.26}$$

Because the weighting functions are linear polynomials of x, this is linear interpolation.

The two examples above have the common form of linear interpolation. We create a variable t that varies from 0 to 1 as we move from data item A to data item B. Intermediate values are just the function $(1 - t)A + tB$. Notice that Equation (2.26) has this form with

$$t = \frac{x - x_i}{x_{i+1} - x_i}.$$

2.7 Triangles

Triangles in both 2D and 3D are the fundamental modeling primitive in many graphics programs. Often information such as color is tagged onto triangle vertices, and this information is interpolated across the triangle. The coordinate system that makes such interpolation straightforward is called *barycentric coordinates*; we will develop these from scratch. We will also discuss 2D triangles, which must be understood before we can draw their pictures on 2D screens.

2.7.1 2D Triangles

If we have a 2D triangle defined by 2D points $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$, we can first find its area:

$$\begin{aligned}\text{area} &= \frac{1}{2}\begin{vmatrix} x_b - x_a & x_c - x_a \\ y_b - y_a & y_c - y_a \end{vmatrix} \\ &= \frac{1}{2}\left(x_a y_b + x_b y_c + x_c y_a - x_a y_c - x_b y_a - x_c y_b\right).\end{aligned} \tag{2.27}$$

The derivation of this formula can be found in Section 5.3. This area will have a positive sign if the points $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ are in counterclockwise order and a negative sign, otherwise.

Often in graphics, we wish to assign a property, such as color, at each triangle vertex and smoothly interpolate the value of that property across the triangle. There are a variety of ways to do this, but the simplest is to use *barycentric* coordinates. One way to think of barycentric coordinates is as a nonorthogonal coordinate system as was discussed briefly in Section 2.4.2. Such a coordinate system is shown in Figure 2.36, where the coordinate origin is $\mathbf{a}$ and the vectors from $\mathbf{a}$ to $\mathbf{b}$ and $\mathbf{c}$ are the basis vectors. With that origin and those basis vectors,

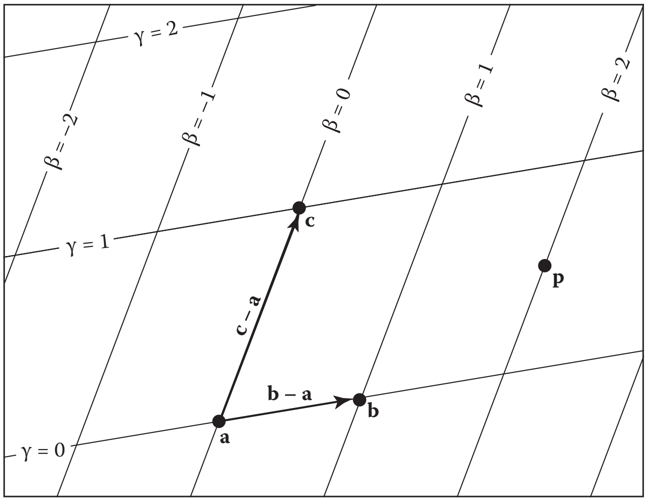


Figure 2.36. A 2D triangle with vertices **a**, **b**, **c** can be used to set up a nonorthogonal coordinate system with origin **a** and basis vectors (**b** – **a**) and (**c** – **a**). A point is then represented by an ordered pair (β, γ). For example, the point **p** = (2.0, 0.5), i.e., **p** = **a** + 2.0 (**b** – **a**) + 0.5 (**c** – **a**).

any point $\mathbf{p}$ can be written as

$$\mathbf{p} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}). \tag{2.28}$$

Note that we can reorder the terms in Equation (2.28) to get

$$\mathbf{p} = (1 - \beta - \gamma)\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}.$$

Often people define a new variable α to improve the symmetry of the equations:

$$\alpha \equiv 1 - \beta - \gamma,$$

which yields the equation

$$\mathbf{p}(\alpha, \beta, \gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}, \tag{2.29}$$

with the constraint that

$$\alpha + \beta + \gamma = 1. \tag{2.30}$$

Barycentric coordinates seem like an abstract and unintuitive construct at first, but they turn out to be powerful and convenient. You may find it useful to think of how street addresses would work in a city where there are two sets of parallel streets, but where those sets are not at right angles. The natural system would essentially be barycentric coordinates, and you would quickly get used to them. Barycentric coordinates are defined for all points on the plane. A particularly nice feature of barycentric coordinates is that a point $\mathbf{p}$ is inside the triangle formed by $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ if and only if

$$\begin{aligned} 0 &< \alpha < 1, \\ 0 &< \beta < 1, \\ 0 &< \gamma < 1. \end{aligned}$$

If one of the coordinates is zero and the other two are between zero and one, then you are on an edge. If two of the coordinates are zero, then the other is one, and you are at a vertex. Another nice property of barycentric coordinates is that Equation (2.29) in effect mixes the coordinates of the three vertices in a smooth way. The same mixing coefficients (α, β, γ) can be used to mix other properties, such as color, as we will see in the next chapter.

Given a point $\mathbf{p}$, how do we compute its barycentric coordinates? One way is to write Equation (2.28) as a linear system with unknowns β and γ, solve, and set $\alpha = 1 - \beta - \gamma$. That linear system is

$$\begin{bmatrix} x_b - x_a & x_c - x_a \\ y_b - y_a & y_c - y_a \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} x_p - x_a \\ y_p - y_a \end{bmatrix}. \tag{2.31}$$

Although it is straightforward to solve Equation (2.31) algebraically, it is often fruitful to compute a direct geometric solution.

One geometric property of barycentric coordinates is that they are the signed scaled distance from the lines through the triangle sides, as is shown for β in Figure 2.37. Recall from Section 2.5.2 that evaluating the equation $f(x, y)$ for the line $f(x, y) = 0$ returns the scaled signed distance from (x, y) to the line. Also recall that if $f(x, y) = 0$ is the equation for a particular line, so is $kf(x, y) = 0$ for any nonzero k. Changing k scales the distance and controls which side of the line has positive signed distance, and which negative. We would like to choose k such that, for example, $kf(x, y) = \beta$. Since k is only one unknown, we can force this with one constraint, namely that at point $\mathbf{b}$ we know $\beta = 1$. So if the line $f_{ac}(x, y) = 0$ goes through both $\mathbf{a}$ and $\mathbf{c}$, then we can compute β for a point (x, y) as follows:

$$\beta = \frac{f_{ac}(x, y)}{f_{ac}(x_b, y_b)}, \tag{2.32}$$

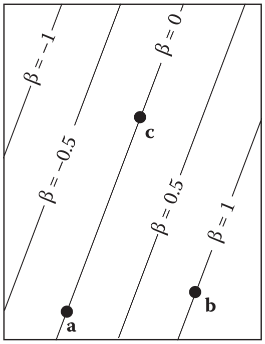


Figure 2.37. The barycentric coordinate β is the signed scaled distance from the line through $\mathbf{a}$ and $\mathbf{c}$.

and we can compute γ and α in a similar fashion. For efficiency, it is usually wise to compute only two of the barycentric coordinates directly and to compute the third using Equation (2.30).

To find this "ideal" form for the line through $\mathbf{p}_0$ and $\mathbf{p}_1$, we can first use the technique of Section 2.5.2 to find *some* valid implicit lines through the vertices. Equation (2.17) gives us

$$f_{ab}(x, y) \equiv (y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a = 0.$$

Note that $f_{ab}(x_c, y_c)$ probably does not equal one, so it is probably not the ideal form we seek. By dividing through by $f_{ab}(x_c, y_c)$ we get

$$\gamma = \frac{(y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a}{(y_a - y_b)x_c + (x_b - x_a)y_c + x_a y_b - x_b y_a}.$$

The presence of the division might worry us because it introduces the possibility of divide-by-zero, but this cannot occur for triangles with areas that are not near zero. There are analogous formulas for α and β, but typically only one is needed:

$$\beta = \frac{(y_a - y_c)x + (x_c - x_a)y + x_a y_c - x_c y_a}{(y_a - y_c)x_b + (x_c - x_a)y_b + x_a y_c - x_c y_a},$$
$$\alpha = 1 - \beta - \gamma.$$

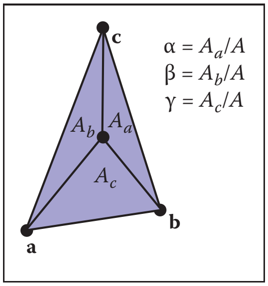


Figure 2.38. The barycentric coordinates are proportional to the areas of the three subtriangles shown.

Another way to compute barycentric coordinates is to compute the areas A_a, A_b, and A_c, of subtriangles as shown in Figure 2.38. Barycentric coordinates obey the rule

$$\begin{aligned}\alpha &= A_a/A,\\ \beta &= A_b/A,\\ \gamma &= A_c/A,\end{aligned} \tag{2.33}$$

where A is the area of the triangle. Note that $A = A_a + A_b + A_c$, so it can be computed with two additions rather than a full area formula. This rule still holds for points outside the triangle if the areas are allowed to be signed. The reason for this is shown in Figure 2.39. Note that these are signed areas and will be computed correctly as long as the same signed area computation is used for both A and the subtriangles A_a, A_b, and A_c.

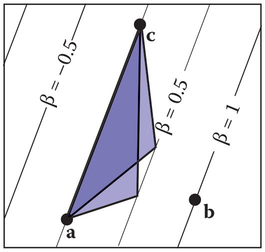


Figure 2.39. The area of the two triangles shown is base times height and are thus the same, as is any triangle with a vertex on the $\beta = 0.5$ line. The height and thus the area is proportional to β.

2.7.2 3D Triangles

One wonderful thing about barycentric coordinates is that they extend almost transparently to 3D. If we assume the points **a**, **b**, and **c** are 3D, then we can still use the representation

$$\mathbf{p} = (1 - \beta - \gamma)\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}.$$

Now, as we vary β and γ, we sweep out a plane.

The normal vector to a triangle can be found by taking the cross product of any two vectors in the plane of the triangle (Figure 2.40). It is easiest to use two of the three edges as these vectors, for example,

$$\mathbf{n} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a}). \tag{2.34}$$

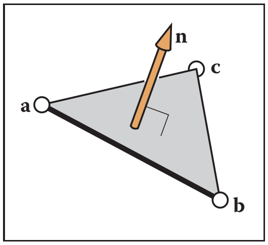


Figure 2.40. The normal vector of the triangle is perpendicular to all vectors in the plane of the triangle, and thus perpendicular to the edges of the triangle.

Note that this normal vector is not necessarily of unit length, and it obeys the right-hand rule of cross products.

The area of the triangle can be found by taking the length of the cross product:

$$\text{area} = \frac{1}{2}\|(\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})\|. \tag{2.35}$$

Note that this is *not* a signed area, so it cannot be used directly to evaluate barycentric coordinates. However, we can observe that a triangle with a "clockwise" vertex order will have a normal vector that points in the opposite direction to the normal of a triangle in the same plane with a "counterclockwise" vertex order. Recall that

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|\ \|\mathbf{b}\|\ \cos\phi,$$

where ϕ is the angle between the vectors. If $\mathbf{a}$ and $\mathbf{b}$ are parallel, then $\cos\phi = \pm 1$, and this gives a test of whether the vectors point in the same or opposite directions. This, along with Equations (2.33), (2.34), and (2.35), suggest the formulas

$$\alpha = \frac{\mathbf{n} \cdot \mathbf{n}_a}{\|\mathbf{n}\|^2},$$

$$\beta = \frac{\mathbf{n} \cdot \mathbf{n}_b}{\|\mathbf{n}\|^2},$$

$$\gamma = \frac{\mathbf{n} \cdot \mathbf{n}_c}{\|\mathbf{n}\|^2},$$

where $\mathbf{n}$ is Equation (2.34) evaluated with vertices $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$; $\mathbf{n}_a$ is Equation (2.34) evaluated with vertices $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{p}$, and so on, i.e.,

$$\begin{aligned} \mathbf{n}_a &= (\mathbf{c} - \mathbf{b}) \times (\mathbf{p} - \mathbf{b}), \\ \mathbf{n}_b &= (\mathbf{a} - \mathbf{c}) \times (\mathbf{p} - \mathbf{c}), \\ \mathbf{n}_c &= (\mathbf{b} - \mathbf{a}) \times (\mathbf{p} - \mathbf{a}). \end{aligned} \tag{2.36}$$

Frequently Asked Questions

• Why isn't there vector division?

It turns out that there is no "nice" analogy of division for vectors. However, it is possible to motivate the quaternions by examining this question in detail (see Hoffmann's book referenced in the chapter notes).

• Is there something as clean as barycentric coordinates for polygons with more than three sides?

Unfortunately there is not. Even convex quadrilaterals are much more complicated. This is one reason triangles are such a common geometric primitive in graphics.

• Is there an implicit form for 3D lines?

No. However, the intersection of two 3D planes defines a 3D line, so a 3D line can be described by two simultaneous implicit 3D equations.

Notes

The history of vector analysis is particularly interesting. It was largely invented by Grassman in the mid-1800s but was ignored and reinvented later (Crowe, 1994). Grassman now has a following in the graphics field of researchers who are developing *Geometric Algebra* based on some of his ideas (Doran & Lasenby, 2003). Readers interested in why the particular scalar and vector products are in some sense the right ones, and why we do not have a commonly used vector division, will find enlightenment in the concise *About Vectors* (Hoffmann, 1975). Another important geometric tool is the *quaternion* invented by Hamilton in the mid-1800s. Quaternions are useful in many situations, but especially where orientations are concerned (Hanson, 2005).

Exercises

1. The *cardinality* of a set is the number of elements it contains. Under IEEE floating point representation (Section 1.5), what is the cardinality of the *floats*?

2. Is it possible to implement a function that maps 32-bit integers to 64-bit integers that has a well-defined inverse? Do all functions from 32-bit integers to 64-bit integers have well-defined inverses?

3. Specify the unit cube (x-, y-, and z-coordinates all between 0 and 1 inclusive) in terms of the Cartesian product of three intervals.

4. If you have access to the natural log function $\ln(x)$, specify how you could use it to implement a $\log(b, x)$ function where b is the base of the log. What should the function do for negative b values? Assume an IEEE floating point implementation.

5. Solve the quadratic equation $2x^2 + 6x + 4 = 0$.

6. Implement a function that takes in coefficients A, B, and C for the quadratic equation $Ax^2 + Bx + C = 0$ and computes the two solutions. Have the function return the number of valid (not NaN) solutions and fill in the return arguments so the smaller of the two solutions is first.

7. Show that the two forms of the quadratic formula on page 17 are equivalent (assuming exact arithmetic) and explain how to choose one for each root in

order to avoid subtracting nearly equal floating point numbers, which leads to loss of precision.

8. Show by counterexample that it is not always true that for 3D vectors $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$, $\mathbf{a} \times (\mathbf{b} \times \mathbf{c}) = (\mathbf{a} \times \mathbf{b}) \times \mathbf{c}$.

9. Given the nonparallel 3D vectors $\mathbf{a}$ and $\mathbf{b}$, compute a right-handed orthonormal basis such that $\mathbf{u}$ is parallel to $\mathbf{a}$ and $\mathbf{v}$ is in the the plane defined by $\mathbf{a}$ and $\mathbf{b}$.

10. What is the gradient of $f(x, y, z) = x^2 + y - 3z^3$?

11. What is a parametric form for the axis-aligned 2D ellipse?

12. What is the implicit equation of the plane through 3D points $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$? What is the parametric equation? What is the normal vector to this plane?

13. Given four 2D points $\mathbf{a}_0$, $\mathbf{a}_1$, $\mathbf{b}_0$, and $\mathbf{b}_1$, design a robust procedure to determine whether the line segments $\mathbf{a}_0\mathbf{a}_1$ and $\mathbf{b}_0\mathbf{b}_1$ intersect.

14. Design a robust procedure to compute the barycentric coordinates of a 2D point with respect to three 2D non-collinear points.

3

Raster Images

Most computer graphics images are presented to the user on some kind of *raster display*. Raster displays show images as rectangular arrays of *pixels*. A common example is a flat-panel computer display or television, which has a rectangular array of small light-emitting pixels that can individually be set to different colors to create any desired image. Different colors are achieved by mixing varying intensities of red, green, and blue light. Most printers, such as laser printers and ink-jet printers, are also raster devices. They are based on scanning: there is no physical grid of pixels, but the image is laid down sequentially by depositing ink at selected points on a grid.

Pixel is short for "picture element."

Color in printers is more complicated, involving mixtures of at least four pigments.

Rasters are also prevalent in input devices for images. A digital camera contains an image sensor comprising a grid of light-sensitive pixels, each of which records the color and intensity of light falling on it. A desktop scanner contains a linear array of pixels that is swept across the page being scanned, making many measurements per second to produce a grid of pixels.

Because rasters are so prevalent in devices, *raster images* are the most common way to store and process images. A raster image is simply a 2D array that stores the *pixel value* for each pixel—usually a color stored as three numbers, for red, green, and blue. A raster image stored in memory can be displayed by using each pixel in the stored image to control the color of one pixel of the display.

Or, maybe it's because raster images are so convenient that raster devices are prevalent.

But we don't always want to display an image this way. We might want to change the size or orientation of the image, correct the colors, or even show the image pasted on a moving three-dimensional surface. Even in televisions, the display rarely has the same number of pixels as the image being displayed. Considerations like these break the direct link between image pixels and display pixels. It's best to think of a raster image as a *device-independent* description of the image to be displayed, and the display device as a way of approximating that ideal image.

There are other ways of describing images besides using arrays of pixels. A *vector image* is described by storing descriptions of shapes—areas of color bounded by lines or curves—with no reference to any particular pixel grid. In essence this amounts to storing the *instructions* for displaying the image rather than the pixels needed to display it. The main advantage of vector images is that they are *resolution independent* and can be displayed well on very high resolution devices. The corresponding disadvantage is that they must be *rasterized* before they can be displayed. Vector images are often used for text, diagrams, mechanical drawings, and other applications where crispness and precision are important and photographic images and complex shading aren't needed.

In this chapter, we discuss the basics of raster images and displays, paying particular attention to the nonlinearities of standard displays. The details of how pixel values relate to light intensities are important to have in mind when we discuss computing images in later chapters.

Or: you have to know what those numbers in your image actually mean.

3.1 Raster Devices

Before discussing raster images in the abstract, it is instructive to look at the basic operation of some specific devices that use these images. A few familiar raster devices can be categorized into a simple hierarchy:

- Output
  - Display
    - Transmissive: liquid crystal display (LCD)
    - Emissive: light-emitting diode (LED) display
  - Hardcopy
    - Binary: ink-jet printer
    - Continuous tone: dye sublimation printer
- Input
  - 2D array sensor: digital camera
  - 1D array sensor: flatbed scanner

3.1.1 Displays

Current displays, including televisions and digital cinematic projectors as well as displays and projectors for computers, are nearly universally based on fixed arrays of pixels. They can be separated into emissive displays, which use pixels that directly emit controllable amounts of light, and transmissive displays, in which the pixels themselves don't emit light but instead vary the amount of light that they allow to pass through them. Transmissive displays require a light source to illuminate them: in a direct-viewed display this is a *backlight* behind the array; in a projector it is a lamp that emits light that is projected onto the screen after passing through the array. An emissive display is its own light source.

Light-emitting diode (LED) displays are an example of the emissive type. Each pixel is composed of one or more LEDs, which are semiconductor devices (based on inorganic or organic semiconductors) that emit light with intensity depending on the electrical current passing through them (see Figure 3.1).

The pixels in a color display are divided into three independently controlled *subpixels*—one red, one green, and one blue—each with its own LED made using different materials so that they emit light of different colors (Figure 3.2). When

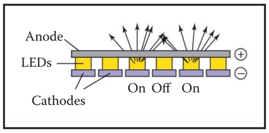


Figure 3.1. The operation of a light-emitting diode (LED) display.

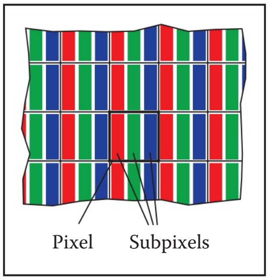


Figure 3.2. The red, green, and blue subpixels within a pixel of a flat-panel display.

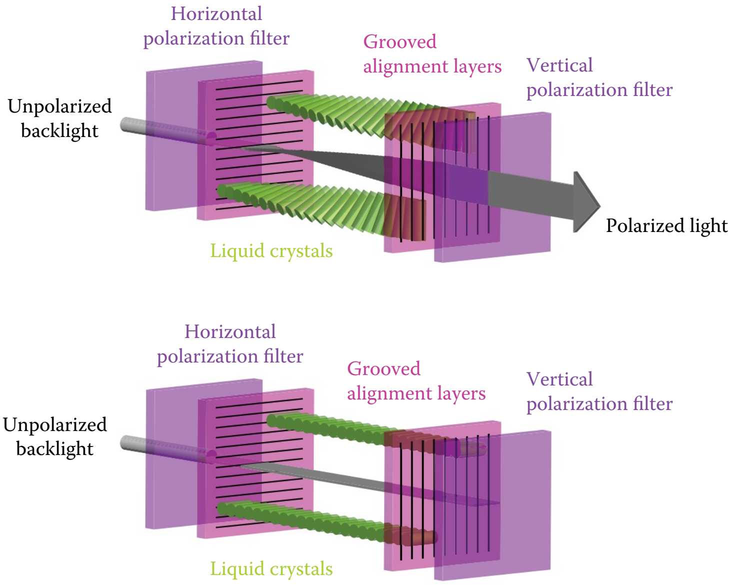


Figure 3.3. One pixel of an LCD display in the off state (bottom), in which the front polarizer blocks all the light that passes the back polarizer, and the on state (top), in which the liquid crystal cell rotates the polarization of the light so that it can pass through the front polarizer. *Figure courtesy Erik Reinhard* (Reinhard, Khan, Akyüz, & Johnson, 2008).

the display is viewed from a distance, the eye can't separate the individual subpixels, and the perceived color is a mixture of red, green, and blue.

Liquid crystal displays (LCDs) are an example of the transmissive type. A liquid crystal is a material whose molecular structure enables it to rotate the polarization of light that passes through it, and the degree of rotation can be adjusted by an applied voltage. An LCD pixel (Figure 3.3) has a layer of polarizing film behind it, so that it is illuminated by polarized light—let's assume it is polarized horizontally.

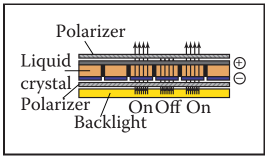


Figure 3.4. The operation of a liquid crystal display (LCD).

A second layer of polarizing film in front of the pixel is oriented to transmit only vertically polarized light. If the applied voltage is set so that the liquid crystal layer in between does not change the polarization, all light is blocked and the pixel is in the "off" (minimum intensity) state. If the voltage is set so that the liquid crystal rotates the polarization by 90 degrees, then all the light that entered through the back of the pixel will escape through the front, and the pixel is fully "on"—it has its maximum intensity. Intermediate voltages will partly rotate the polarization so that the front polarizer partly blocks the light, resulting in intensities between the minimum and maximum (Figure 3.4). Like color LED displays, color LCDs have red, green, and blue subpixels within each pixel, which are three independent pixels with red, green, and blue color filters over them.

Any type of display with a fixed pixel grid, including these and other technologies, has a fundamentally fixed *resolution* determined by the size of the grid. For displays and images, resolution simply means the dimensions of the pixel grid: if a desktop monitor has a resolution of 1920 × 1200 pixels, this means that it has 2,304,000 pixels arranged in 1920 columns and 1200 rows.

The resolution of a display is sometimes called its "native resolution" since most displays can handle images of other resolutions, via built-in conversion.

An image of a different resolution, to fill the screen, must be converted into a 1920 × 1200 image using the methods of Chapter 9.

3.1.2 Hardcopy Devices

The process of recording images permanently on paper has very different constraints from showing images transiently on a display. In printing, pigments are distributed on paper or another medium so that when light reflects from the paper it forms the desired image. Printers are raster devices like displays, but many printers can only print *binary images*—pigment is either deposited or not at each grid position, with no intermediate amounts possible.

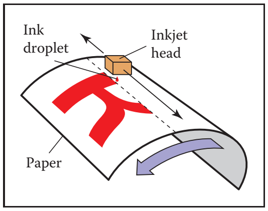


Figure 3.5. The operation of an ink-jet printer.

An ink-jet printer (Figure 3.5) is an example of a device that forms a raster image by scanning. An ink-jet print head contains liquid ink carrying pigment, which can be sprayed in very small drops under electronic control. The head moves across the paper, and drops are emitted as it passes grid positions that should receive ink; no ink is emitted in areas intended to remain blank. After each sweep the paper is advanced slightly, and then the next row of the grid is laid down. Color prints are made by using several print heads, each spraying ink with a different pigment, so that each grid position can receive any combination of different colored drops. Because all drops are the same, an ink-jet printer prints binary images: at each grid point there is a drop or no drop; there are no intermediate shades.

There are also continuous ink-jet printers that print in a continuous helical path on paper wrapped around a spinning drum, rather than moving the head back and forth.

An ink-jet printer has no physical array of pixels; the resolution is determined by how small the drops can be made and how far the paper is advanced after each sweep. Many ink-jet printers have multiple nozzles in the print head, enabling several sweeps to be made in one pass, but it is the paper advance, not the nozzle spacing, that ultimately determines the spacing of the rows.

The *thermal dye transfer* process is an example of a *continuous tone* printing process, meaning that varying amounts of dye can be deposited at each pixel—it is not all-or-nothing like an ink-jet printer (Figure 3.6). A *donor ribbon* containing colored dye is pressed between the paper, or *dye receiver*, and a *print head* containing a linear array of heating elements, one for each column of pixels in the image. As the paper and ribbon move past the head, the heating elements switch on and off to heat the ribbon in areas where dye is desired, causing the dye to diffuse from the ribbon to the paper. This process is repeated for each of several dye colors. Since higher temperatures cause more dye to be transferred, the amount of each dye deposited at each grid position can be controlled, allowing a continuous range of colors to be produced. The number of heating elements in the print head establishes a fixed resolution in the direction across the page, but the resolution along the page is determined by the rate of heating and cooling compared to the speed of the paper.

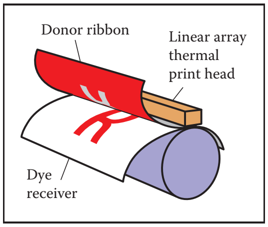


Figure 3.6. The operation of a thermal dye transfer printer.

Unlike displays, the resolution of printers is described in terms of the *pixel density* instead of the total count of pixels. So a thermal dye transfer printer that has elements spaced 300 per inch across its print head has a resolution of 300 *pixels per inch* (ppi) across the page. If the resolution along the page is chosen to be the same, we can simply say the printer's resolution is 300 ppi. An ink-jet printer that places dots on a grid with 1200 grid points per inch is described as having a resolution of 1200 *dots per inch* (dpi). Because the ink-jet printer is a binary device, it requires a much finer grid for at least two reasons. Because edges are abrupt black/white boundaries, very high resolution is required to avoid stair-stepping, or aliasing, from appearing (see Section 8.3). When continuous-tone images are printed, the high resolution is required to simulate intermediate colors by printing varying-density dot patterns called *halftones*.

The term "dpi" is all too often used to mean "pixels per inch," but dpi should be used in reference to binary devices and ppi in reference to continuous-tone devices.

3.1.3 Input Devices

Raster images have to come from somewhere, and any image that wasn't computed by some algorithm has to have been measured by some *raster input device*, most often a camera or scanner. Even in rendering images of 3D scenes, photographs are used constantly as texture maps (see Chapter 11). A raster input device has to make a light measurement for each pixel, and (like output devices) they are usually based on arrays of sensors.

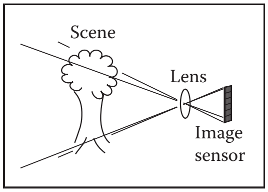
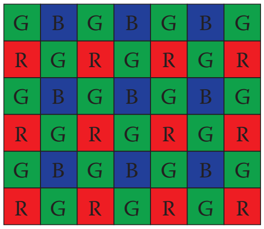


Figure 3.7. The operation of a digital camera.

A digital camera is an example of a 2D array input device. The image sensor in a camera is a semiconductor device with a grid of light-sensitive pixels. Two common types of arrays are known as CCDs (charge-coupled devices) and CMOS (complimentary metal–oxide–semiconductor) image sensors. The camera's lens projects an image of the scene to be photographed onto the sensor, and then each pixel measures the light energy falling on it, ultimately resulting in a number that goes into the output image (Figure 3.7). In much the same way as color displays use red, green, and blue subpixels, most color cameras work by using a *color-filter array* or *mosaic* to allow each pixel to see only red, green, or blue light, leaving the image processing software to fill in the missing values in a process known as *demosaicking* (Figure 3.8).



G	B	G	B	G	B	G
R	G	R	G	R	G	R
G	B	G	B	G	B	G
R	G	R	G	R	G	R
G	B	G	B	G	B	G
R	G	R	G	R	G	R

Figure 3.8. Most color digital cameras use a color-filter array similar to the *Bayer mosaic* shown here. Each pixel measures either red, green, or blue light.

Other cameras use three separate arrays, or three separate layers in the array, to measure independent red, green, and blue values at each pixel, producing a usable color image without further processing. The resolution of a camera is determined by the fixed number of pixels in the array and is usually quoted using the total count of pixels: a camera with an array of 3000 columns and 2000 rows produces an image of resolution 3000 × 2000, which has 6 million pixels, and is called a 6 megapixel (MP) camera. It's important to remember that a mosiac sensor does not measure a complete color image, so a camera that measures the same number of pixels but with independent red, green, and blue measurements records more information about the image than one with a mosaic sensor.

People who are selling cameras use "mega" to mean 10^6, not 2^{20} as with megabytes.

A flatbed scanner also measures red, green, and blue values for each of a grid of pixels, but like a thermal dye transfer printer it uses a 1D array that sweeps across the page being scanned, making many measurements per second. The resolution across the page is fixed by the size of the array, and the resolution along the page is determined by the frequency of measurements compared to the speed at which the scan head moves. A color scanner has a $3 \times n_x$ array, where n_x is the number of pixels across the page, with the three rows covered by red, green, and blue filters. With an appropriate delay between the times at which the three colors are measured, this allows three independent color measurements at each grid point. As with continuous-tone printers, the resolution of scanners is reported in pixels per inch (ppi).

The resolution of a scanner is sometimes called its "optical resolution" since most scanners can produce images of other resolutions, via built-in conversion.

With this concrete information about where our images come from and where they will go, we'll now discuss images more abstractly, in the way we'll use them in graphics algorithms.

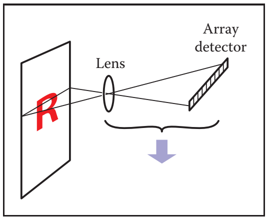


Figure 3.9. The operation of a flatbed scanner.

3.2 Images, Pixels, and Geometry

We know that a raster image is a big array of pixels, each of which stores information about the color of the image at its grid point. We've seen what various output devices do with images we send to them and how input devices derive them from images formed by light in the physical world. But for computations in the computer, we need a convenient abstraction that is independent of the specifics of any device, that we can use to reason about how to produce or interpret the values stored in images.

When we measure or reproduce images, they take the form of two-dimensional distributions of light energy: the light emitted from the monitor as a function of position on the face of the display; the light falling on a camera's image sensor as a function of position across the sensor's plane; the *reflectance*, or fraction of light reflected (as opposed to absorbed) as a function of position on a piece of paper. So in the physical world, images are functions defined over two-dimensional areas—almost always rectangles. So we can abstract an image as a function

"A pixel is not a little square!"
—Alvy Ray Smith
(A. R. Smith, 1995)

$$I(x, y) : R \to V,$$

where $R \subset \mathbb{R}^2$ is a rectangular area and V is the set of possible pixel values. The simplest case is an idealized grayscale image where each point in the rectangle has just a brightness (no color), and we can say $V = \mathbb{R}^+$ (the nonnegative reals). An idealized color image, with red, green, and blue values at each pixel, has $V = (\mathbb{R}^+)^3$. We'll discuss other possibilities for V in the next section.

Are there any raster devices that are not rectangular?

How does a raster image relate to this abstract notion of a continuous image? Looking to the concrete examples, a pixel from a camera or scanner is a measurement of the average color of the image over some small area around the pixel. A display pixel, with its red, green, and blue subpixels, is designed so that the average color of the image over the face of the pixel is controlled by the corresponding pixel value in the raster image. In both cases, the pixel value is a local average of the color of the image, and it is called a *point sample* of the image. In other words, when we find the value x in a pixel, it means "the value of the image in the vicinity of this grid point is x." The idea of images as sampled representations of functions is explored further in Chapter 9.

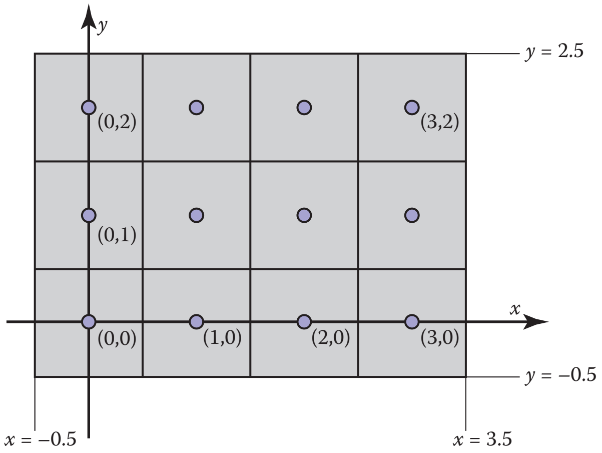


Figure 3.10. Coordinates of a four pixel × three pixel screen. Note that in some APIs the *y*-axis will point downward.

A mundane but important question is where the pixels are located in 2D space. This is only a matter of convention, but establishing a consistent convention is important! In this book, a raster image is indexed by the pair (i, j) indicating the column (i) and row (j) of the pixel, counting from the bottom left. If an image has n_x columns and n_y rows of pixels, the bottom-left pixel is $(0, 0)$ and the top-right is pixel $(n_x - 1, n_y - 1)$. We need 2D real screen coordinates to specify pixel positions. We will place the pixels' sample points at integer coordinates, as shown by the 4 × 3 screen in Figure 3.10.

In some APIs, and many file formats, the rows of an image are organized top-to-bottom, so that (0, 0) is at the top left. This is for historical reasons: the rows in analog television transmission started from the top.

The rectangular domain of the image has width n_x and height n_y and is centered on this grid, meaning that it extends half a pixel beyond the last sample point on each side. So the rectangular domain of a $n_x \times n_y$ image is

$$R = [-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5] .$$

Some systems shift the coordinates by half a pixel to place the sample points halfway between the integers but place the edges of the image at integers.

Again, these coordinates are simply conventions, but they will be important to remember later when implementing cameras and viewing transformations.

3.2.1 Pixel Values

So far we have described the values of pixels in terms of real numbers, representing intensity (possibly separately for red, green, and blue) at a point in the image. This suggests that images should be arrays of floating-point numbers, with either one (for *grayscale*, or black and white, images) or three (for RGB color images)

32-bit floating point numbers stored per pixel. This format is sometimes used, when its precision and range of values are needed, but images have a lot of pixels and memory and bandwidth for storing and transmitting images are invariably scarce. Just one ten-megapixel photograph would consume about 115 MB of RAM in this format.

Why 115 MB and not 120 MB?

Less range is required for images that are meant to be displayed directly. While the range of possible light intensities is unbounded in principle, any given device has a decidedly finite maximum intensity, so in many contexts it is perfectly sufficient for pixels to have a bounded range, usually taken to be $[0, 1]$ for simplicity. For instance, the possible values in an 8-bit image are $0, 1/255, 2/255, \ldots, 254/255, 1$. Images stored with floating-point numbers, allowing a wide range of values, are often called *high dynamic range* (HDR) images to distinguish them from fixed-range, or *low dynamic range* (LDR) images that are stored with integers. See Chapter 21 for an in-depth discussion of techniques and applications for high dynamic range images.

The denominator of 255, rather than 256, is awkward, but being able to represent 0 and 1 exactly is important.

Here are some pixel formats with typical applications:

- 1-bit grayscale—text and other images where intermediate grays are not desired (high resolution required);
- 8-bit RGB fixed-range color (24 bits total per pixel)—web and email applications, consumer photographs;
- 8- or 10-bit fixed-range RGB (24–30 bits/pixel)—digital interfaces to computer displays;
- 12- to 14-bit fixed-range RGB (36–42 bits/pixel)—raw camera images for professional photography;
- 16-bit fixed-range RGB (48 bits/pixel)—professional photography and printing; intermediate format for image processing of fixed-range images;
- 16-bit fixed-range grayscale (16 bits/pixel)—radiology and medical imaging;
- 16-bit "half-precision" floating-point RGB—HDR images; intermediate format for real-time rendering;
- 32-bit floating-point RGB—general-purpose intermediate format for software rendering and processing of HDR images.

Reducing the number of bits used to store each pixel leads to two distinctive types of *artifacts*, or artificially introduced flaws, in images. First, encoding images with fixed-range values produces *clipping* when pixels that would otherwise be brighter than the maximum value are set, or clipped, to the maximum representable value. For instance, a photograph of a sunny scene may include reflections that are much brighter than white surfaces; these will be clipped (even if they were measured by the camera) when the image is converted to a fixed range to be displayed. Second, encoding images with limited precision leads to *quantization* artifacts, or *banding*, when the need to round pixel values to the nearest representable value introduces visible jumps in intensity or color. Banding can be particularly insidious in animation and video, where the bands may not be objectionable in still images, but become very visible when they move back and forth.

3.2.2 Monitor Intensities and Gamma

All modern monitors take digital input for the “value” of a pixel and convert this to an intensity level. Real monitors have some nonzero intensity when they are off because the screen reflects some light. For our purposes we can consider this “black” and the monitor fully on as “white.” We assume a numeric description of pixel color that ranges from zero to one. Black is zero, white is one, and a gray halfway between black and white is 0.5. Note that here “halfway” refers to the physical amount of light coming from the pixel, rather than the appearance. The human perception of intensity is nonlinear and will not be part of the present discussion; see Chapter 20 for more.

There are two key issues that must be understood to produce correct images on monitors. The first is that monitors are nonlinear with respect to input. For example, if you give a monitor 0, 0.5, and 1.0 as inputs for three pixels, the intensities displayed might be 0, 0.25, and 1.0 (off, one-quarter fully on, and fully on). As an approximate characterization of this nonlinearity, monitors are commonly characterized by a γ (“gamma”) value. This value is the degree of freedom in the formula

$$\text{displayed intensity} = (\text{maximum intensity})a^{\gamma}, \tag{3.1}$$

where a is the input pixel value between zero and one. For example, if a monitor has a gamma of 2.0, and we input a value of $a = 0.5$, the displayed intensity will be one fourth the maximum possible intensity because $0.5^2 = 0.25$. Note that $a = 0$ maps to zero intensity and $a = 1$ maps to the maximum intensity regardless of the value of γ. Describing a display’s nonlinearity using γ is only an approximation; we do not need a great deal of accuracy in estimating the γ of a device. A nice visual way to gauge the nonlinearity is to find what value of a gives an intensity halfway between black and white. This a will be

$$0.5 = a^{\gamma}.$$

If we can find that a, we can deduce γ by taking logarithms on both sides:

$$\gamma = \frac{\ln 0.5}{\ln a}.$$

We can find this a by a standard technique where we display a checkerboard pattern of black and white pixels next to a square of gray pixels with input a (Figure 3.11), then ask the user to adjust a (with a slider, for instance) until the two sides match in average brightness. When you look at this image from a distance (or without glasses if you are nearsighted), the two sides of the image will look about the same when a is producing an intensity halfway between black and white. This is because the blurred checkerboard is mixing even numbers of white and black pixels so the overall effect is a uniform color halfway between white and black.

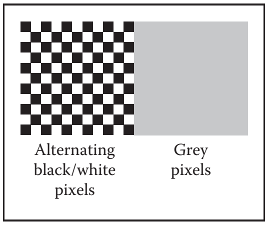


Figure 3.11. Alternating black and white pixels viewed from a distance are halfway between black and white. The gamma of a monitor can be inferred by finding a gray value that appears to have the same intensity as the black and white pattern.

Once we know γ, we can *gamma correct* our input so that a value of $a = 0.5$ is displayed with intensity halfway between black and white. This is done with the transformation

$$a' = a^{\frac{1}{\gamma}}.$$

When this formula is plugged into Equation (3.1) we get

$$\begin{aligned} \text{displayed intensity} &= (a')^{\gamma} = \left(a^{\frac{1}{\gamma}}\right)^{\gamma} (\text{maximum intensity}) \\ &= a(\text{maximum intensity}). \end{aligned}$$

For monitors with analog interfaces, which have difficulty changing intensity rapidly along the horizontal direction, horizontal black and white stripes work better than a checkerboard.

Another important characteristic of real displays is that they take quantized input values. So while we can manipulate intensities in the floating point range $[0, 1]$, the detailed input to a monitor is a fixed-size integer. The most common range for this integer is 0–255 which can be held in 8 bits of storage. This means that the possible values for a are not any number in $[0, 1]$ but instead

$$\text{possible values for } a = \left\{ \frac{0}{255}, \frac{1}{255}, \frac{2}{255}, \dots, \frac{254}{255}, \frac{255}{255} \right\}.$$

This means the possible displayed intensity values are approximately

$$\left\{ M\left(\frac{0}{255}\right)^{\gamma}, M\left(\frac{1}{255}\right)^{\gamma}, M\left(\frac{2}{255}\right)^{\gamma}, \dots, M\left(\frac{254}{255}\right)^{\gamma}, M\left(\frac{255}{255}\right)^{\gamma} \right\},$$

where M is the maximum intensity. In applications where the exact intensities need to be controlled, we would have to actually measure the 256 possible intensities, and these intensities might be different at different points on the screen, especially for CRTs. They might also vary with viewing angle. Fortunately, few applications require such accurate calibration.

3.3 RGB Color

In grade school you probably learned that the primaries are red, yellow, and blue, and that, e.g., yellow + blue = green. This is *subtractive* color mixing, which is fundamentally different from the more familiar additive mixing that happens in displays.

Most computer graphics images are defined in terms of red-green-blue (RGB) color. RGB color is a simple space that allows straightforward conversion to the controls for most computer screens. In this section, RGB color is discussed from a user's perspective, and operational facility is the goal. A more thorough discussion of color is given in Chapter 19, but the mechanics of RGB color space will allow us to write most graphics programs. The basic idea of RGB color space is that the color is displayed by mixing three *primary* lights: one red, one green, and one blue. The lights mix in an *additive* manner.

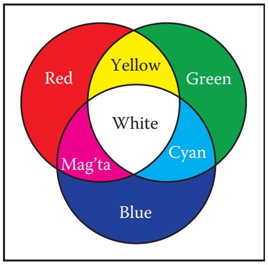


Figure 3.12. The additive mixing rules for colors red/-green/blue.

In RGB additive color mixing we have (Figure 3.12)

$$\begin{aligned}\text{red} + \text{green} &= \text{yellow},\\ \text{green} + \text{blue} &= \text{cyan},\\ \text{blue} + \text{red} &= \text{magenta},\\ \text{red} + \text{green} + \text{blue} &= \text{white}.\end{aligned}$$

The color "cyan" is a blue-green, and the color "magenta" is a purple.

If we are allowed to dim the primary lights from fully off (indicated by pixel value 0) to fully on (indicated by 1), we can create all the colors that can be displayed on an RGB monitor. The red, green, and blue pixel values create a three-dimensional *RGB color cube* that has a red, a green, and a blue axis. Allowable coordinates for the axes range from zero to one. The color cube is shown graphically in Figure 3.13.

The colors at the corners of the cube are

$$\begin{aligned}\text{black} &= (0,0,0),\\ \text{red} &= (1,0,0),\\ \text{green} &= (0,1,0),\\ \text{blue} &= (0,0,1),\\ \text{yellow} &= (1,1,0),\\ \text{magenta} &= (1,0,1),\end{aligned}$$

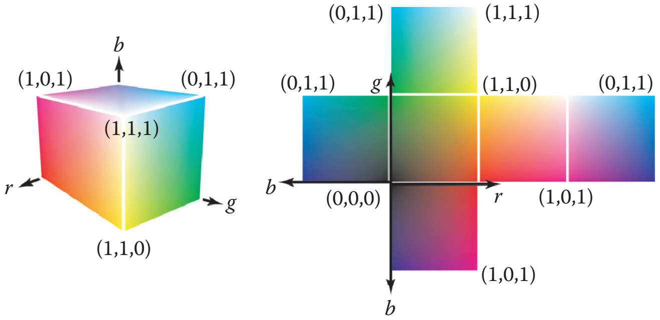


Figure 3.13. The RGB color cube in 3D and its faces unfolded. Any RGB color is a point in the cube.

$$\begin{aligned} \text{cyan} &= (0,1,1), \\ \text{white} &= (1,1,1). \end{aligned}$$

Actual RGB levels are often given in quantized form, just like the grayscales discussed in Section 3.2.2. Each component is specified with an integer. The most common size for these integers is one byte each, so each of the three RGB components is an integer between 0 and 255. The three integers together take up three bytes, which is 24 bits. Thus a system that has "24-bit color" has 256 possible levels for each of the three primary colors. Issues of gamma correction discussed in Section 3.2.2 also apply to each RGB component separately.

3.4 Alpha Compositing

Often we would like to only partially overwrite the contents of a pixel. A common example of this occurs in *compositing*, where we have a background and want to insert a foreground image over it. For opaque pixels in the foreground, we just replace the background pixel. For entirely transparent foreground pixels, we do not change the background pixel. For *partially* transparent pixels, some care must be taken. Partially transparent pixels can occur when the foreground object has partially transparent regions, such as glass. But, the most frequent case where foreground and background must be blended is when the foreground object only partly covers the pixel, either at the edge of the foreground object, or when there are sub-pixel holes such as between the leaves of a distant tree.

The most important piece of information needed to blend a foreground object over a background object is the *pixel coverage*, which tells the fraction of the pixel covered by the foreground layer. We can call this fraction α. If we want to composite a foreground color $\mathbf{c}_f$ over background color $\mathbf{c}_b$, and the fraction of the pixel covered by the foreground is α, then we can use the formula

$$\mathbf{c} = \alpha \mathbf{c}_f + (1 - \alpha)\mathbf{c}_b. \tag{3.2}$$

For an opaque foreground layer, the interpretation is that the foreground object covers area α within the pixel's rectangle and the background object covers the remaining area, which is $(1 - \alpha)$. For a transparent layer (think of an image painted on glass or on tracing paper, using translucent paint), the interpretation is that the foreground layer blocks the fraction $(1 - \alpha)$ of the light coming through from the background and contributes a fraction α of its own color to replace what was removed. An example of using Equation (3.2) is shown in Figure 3.14.

Since the weights of the foreground and background layers add up to 1, the color won't change if the foreground and background layers have the same color.

The α values for all the pixels in an image might be stored in a separate grayscale image, which is then known as an *alpha mask* or *transparency mask*. Or the information can be stored as a fourth channel in an RGB image, in which case it is called the *alpha channel*, and the image can be called an RGBA image. With 8-bit images, each pixel then takes up 32 bits, which is a conveniently sized chunk in many computer architectures.

Although Equation (3.2) is what is usually used, there are a variety of situations where α is used differently (Porter & Duff, 1984).

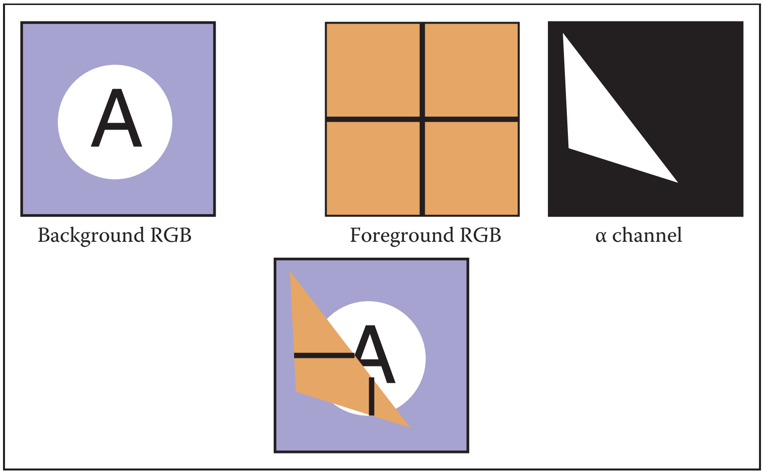


Figure 3.14. An example of compositing using Equation (3.2). The foreground image is in effect cropped by the α channel before being put on top of the background image. The resulting composite is shown on the bottom.

3.4.1 Image Storage

Most RGB image formats use eight bits for each of the red, green, and blue channels. This results in approximately three megabytes of raw information for a single million-pixel image. To reduce the storage requirement, most image formats allow for some kind of compression. At a high level, such compression is either *lossless* or *lossy*. No information is discarded in lossless compression, while some information is lost unrecoverably in a lossy system. Popular image storage formats include:

- jpeg. This lossy format compresses image blocks based on thresholds in the human visual system. This format works well for natural images.
- tiff. This format is most commonly used to hold binary images or losslessly compressed 8- or 16-bit RGB although many other options exist.
- ppm. This very simple lossless, uncompressed format is most often used for 8-bit RGB images although many options exist.
- png. This is a set of lossless formats with a good set of open source management tools.

Because of compression and variants, writing input/output routines for images can be involved. Fortunately one can usually rely on library routines to read and write standard file formats. For quick-and-dirty applications, where simplicity is valued above efficiency, a simple choice is to use raw ppm files, which can often be written simply by dumping the array that stores the image in memory to a file, prepending the appropriate header.

Frequently Asked Questions

• Why don't they just make monitors linear and avoid all this gamma business?

Ideally the 256 possible intensities of a monitor should *look* evenly spaced as opposed to being linearly spaced in energy. Because human perception of intensity is itself nonlinear, a gamma between 1.5 and 3 (depending on viewing conditions) will make the intensities approximately uniform in a subjective sense. In this way, gamma is a feature. Otherwise the manufacturers would make the monitors linear.

Exercise

1. Simulate an image acquired from the Bayer mosaic by taking a natural image (preferably a scanned photo rather than a digital photo where the Bayer mosaic may already have been applied) and creating a grayscale image composed of interleaved red/green/blue channels. This simulates the raw output of a digital camera. Now create a true RGB image from that output and compare with the original.

4

Ray Tracing

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

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

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

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

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

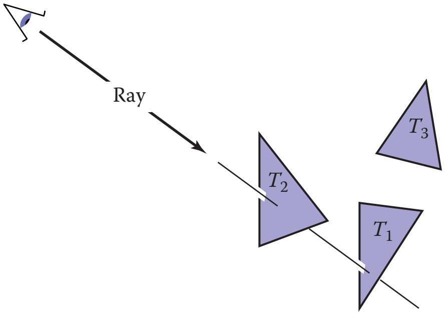


Figure 4.1. The ray is "traced" into the scene and the first object hit is the one seen through the pixel. In this case, the triangle T_2 is returned.

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

4.1 The Basic Ray-Tracing Algorithm

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

A basic ray tracer therefore has three parts:

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

The structure of the basic ray tracing program is:

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

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

4.2 Perspective

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

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

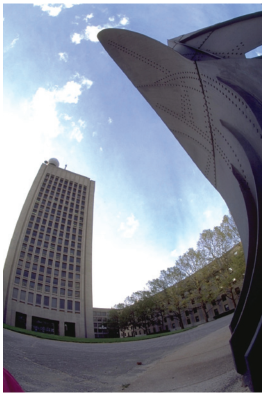


Figure 4.2. An image taken with a fisheye lens is not a linear perspective image. *Photo courtesy Philip Greenspun.*

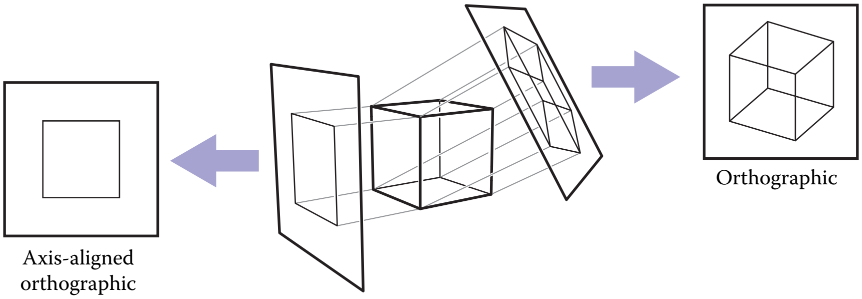


Figure 4.3. When projection lines are parallel and perpendicular to the image plane, the resulting views are called orthographic.

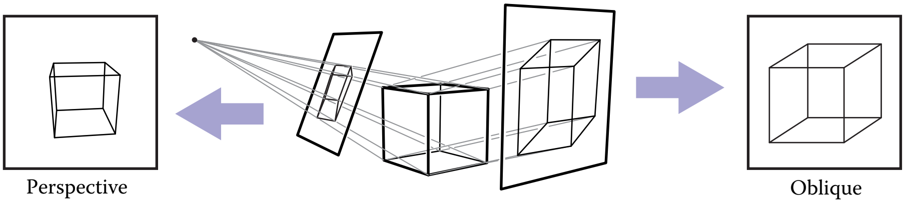


Figure 4.4. A parallel projection that has the image plane at an angle to the projection direction is called oblique (right). In perspective projection, the projection lines all pass through the viewpoint, rather than being parallel (left). The illustrated perspective view is non-oblique because a projection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

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

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

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

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

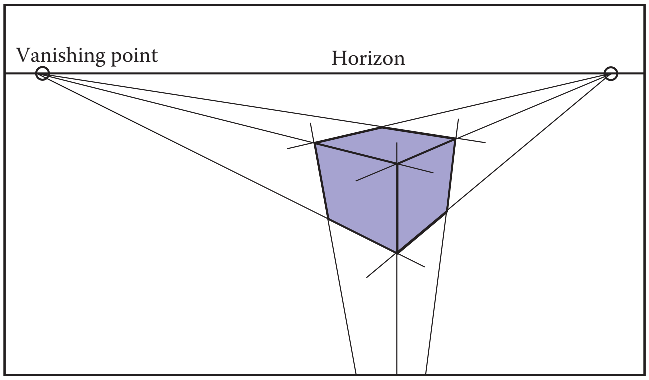


Figure 4.5. In three-point perspective, an artist picks "vanishing points" where parallel lines meet. Parallel horizontal lines will meet at a point on the horizon. Every set of parallel lines has its own vanishing points. These rules are followed automatically if we implement perspective based on the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

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

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

4.3 Computing Viewing Rays

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

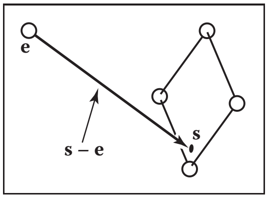


Figure 4.6. The ray from the eye to a point on the image plane.

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

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

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

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

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

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

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

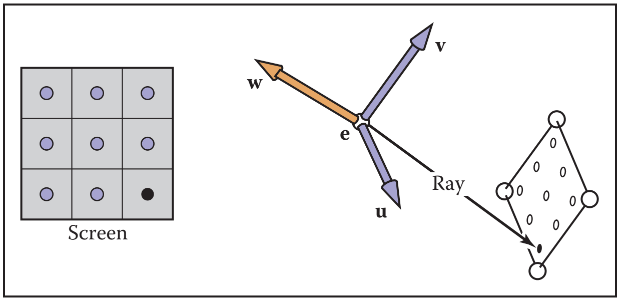


Figure 4.7. The sample points on the screen are mapped to a similar array on the 3D window. A viewing ray is sent to each of these locations.

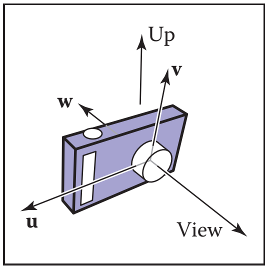


Figure 4.8. The vectors of the camera frame, together with the view direction and up direction. The **w** vector is opposite the view direction, and the **v** vector is coplanar with **w** and the up vector.

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.

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."

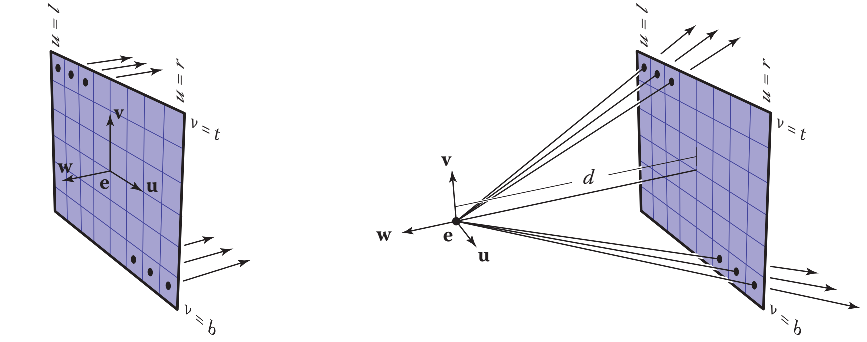
It might seem logical that orthographic viewing rays should start from infinitely far away, but then it would not be possible to make orthographic views of an object inside a room, for instance.

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.)

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

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

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



Parallel projection
same direction, different origins

Perspective projection
same origin, different directions

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

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

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

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

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

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

4.3.2 Perspective Views

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

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

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

4.4 Ray-Object Intersection

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

4.4.1 Ray-Sphere Intersection

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

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

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

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

We can write this same equation in vector form:

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

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

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

Rearranging terms yields

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

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

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

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

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

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

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

4.4.2 Ray-Triangle Intersection

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

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

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

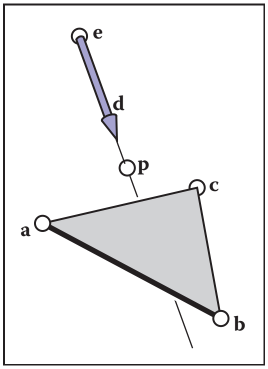


Figure 4.10. The ray hits the plane containing the triangle at point **p**.

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 **a**, **b**, and **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 **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 **r**, vector3 **a**, vector3 **b**, vector3 **c**,
               interval $[t_0, t_1]$)
compute $t$
**if** $(t < t_0)$ or $(t > t_1)$ **then**
   **return** false
compute $\gamma$
**if** $(\gamma < 0)$ or $(\gamma > 1)$ **then**
   **return** false
compute $\beta$
**if** $(\beta < 0)$ or $(\beta > 1 - \gamma)$ **then**
   **return** false
**return** true
```

4.4.3 Ray-Polygon Intersection

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

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

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

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

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

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

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

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

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

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

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

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

4.4.4 Intersecting a Group of Objects

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

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

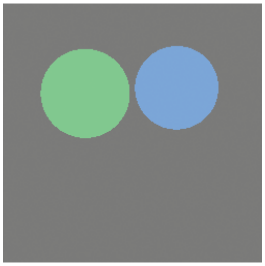


Figure 4.11. A simple scene rendered with only ray generation and surface intersection, but no shading; each pixel is just set to a fixed color depending on which object it hit.

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 toward the light source; the view direction $\mathbf{v}$, which is a unit vector pointing toward the eye or camera; the surface normal $\mathbf{n}$, which is a unit vector perpendicular to the surface at the point where reflection is taking place; and the characteristics of the surface—color, shininess, or other properties depending on the particular model.

4.5.1 Lambertian Shading

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

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

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

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

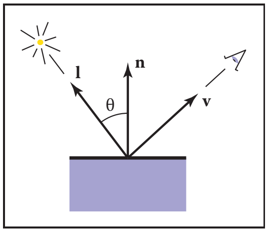


Figure 4.12. Geometry for Lambertian shading.

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

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

4.5.2 Blinn-Phong Shading

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

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

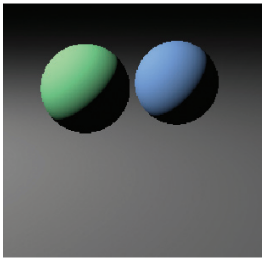


Figure 4.13. A simple scene rendered with diffuse shading from a single light source.

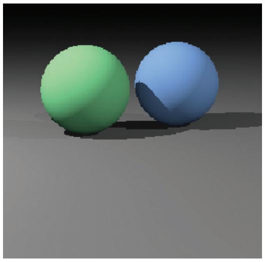


Figure 4.14. A simple scene rendered with diffuse shading and shadows (Section 4.7) from three light sources.

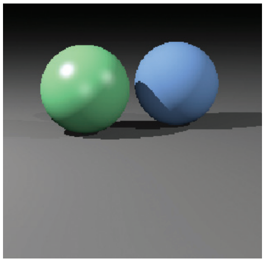


Figure 4.15. A simple scene rendered with diffuse shading (blue sphere), Blinn-Phong shading (green sphere), and shadows from three light sources.

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

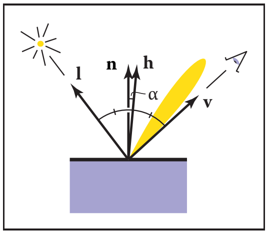


Figure 4.16. Geometry for Blinn-Phong shading.

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

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

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

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

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

4.5.3 Ambient Shading

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

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

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

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

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

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

4.5.4 Multiple Point Lights

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

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

where I_i, $\mathbf{l}_i$, and $\mathbf{h}_i$ are the intensity, direction, and half vector of the 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)
```