Fundamentals of Ray Tracing

Contents

Chapter 1

Introduction

This text and the included C++ source code provide a demonstration of a 3D graphics technique called *ray tracing*. Ray tracing mathematically simulates the behavior of light and its interactions with physical objects — reflection, refraction, and shadows. Sophisticated ray tracing techniques make possible realistic special effects for movies and games, as well as helpful visualizations of organic molecules for biologists and machinery designs for mechanical engineers. The field of ray tracing has grown extremely complex over the past three decades, as everyday computers have grown exponentially in speed and memory capacity.

My goal in this tutorial is to provide a starting point that omits as much complexity as possible, and assumes only that the reader has a good understanding of algebra and analytic geometry. I include a review chapter on vectors for those who need it; vectors are of primary importance in ray tracing. Some trigonometry will be helpful at times, but only in small doses, and the necessary parts will be explained. To follow the programming examples, the reader must also understand the C++ programming language. By following along with this text and the C++ code that accompanies it, you will understand core concepts of computer graphics sufficiently to build your own mathematical models and see them transformed into 3D images.

To keep things simple, I will limit the covered topics in certain ways. The shapes we will draw will be simple geometric solids like spheres, cubes, and cylinders, not complex artistic endeavors like people or animals. (However, we will be able to combine the simple shapes in interesting and even surprising ways within a single image.) All light sources will be modeled as perfect points, resulting in unnaturally crisp shadows — we thus avoid a tremendous amount of complexity for diffuse light sources and mutual reflections between objects in the scene.

In spite of some limitations, the source code provided with this book has quite a bit of flexibility in its modeling of optics. Objects can combine glossy reflection, diffuse scattering, and transparent refraction. Ray-traced images can thus possess visual appeal from realistic shading, perspective, lensing, and sense of depth, as shown in Figures 1.1 and 1.2. By default, an object has uniform glossiness, opacity, and color across its surface, but the programmer has the ability to derive custom C++ classes that override this behavior, allowing a single object to have varying features across its surface.

To keep distractions to a minimum, I will limit the programming examples to the command line. I want to present ideas and algorithms that you can experiment with whether you are using Microsoft Windows, Mac OS X, or Linux. All three operating systems have very different and complicated programming models for their respective graphical user interfaces (GUIs). But they all have a very simple and similar way of writing command line programs. Instead of trying to render images in a GUI window, the source code accompanying this book generates an image, saves it to an output file, and exits. After the program has finished, you can use an image viewer or web browser to look at the output file. You can then modify the source code, build and run the program again, and see how the output changed.

Overall, my goal is to make the fundamentals of ray tracing understandable. There are many places in this book where an expert on the subject could fairly say, "there is a faster way to do that" or "a more sophisticated approach is possible," but in every case where I have had to make a choice, I have leaned toward making this as gentle an introduction as possible to computer graphics. In some cases, I have left out certain optical subtleties needed to make completely photo-realistic images, and in other cases I have implemented an algorithm in a

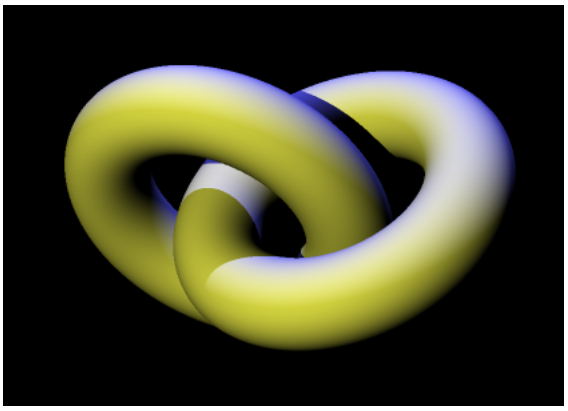

Figure 1.1: A ray-traced image.

slower than ideal way; but in every such case, the resulting C++ code is smaller and simpler to read and digest. My hope is that this book therefore provides a helpful bridge to newcomers to the field of ray tracing who want to understand more than vague concepts but are put off by the overwhelming complexity of academic journals and postgraduate-level textbooks. In this regard, I will consider *Fundamentals of Ray Tracing* a success if readers find that it satisfies their curiosity without frustrating them, and if some are intrigued into pursuing more advanced treatments of the subject that otherwise would have been too intimidating.

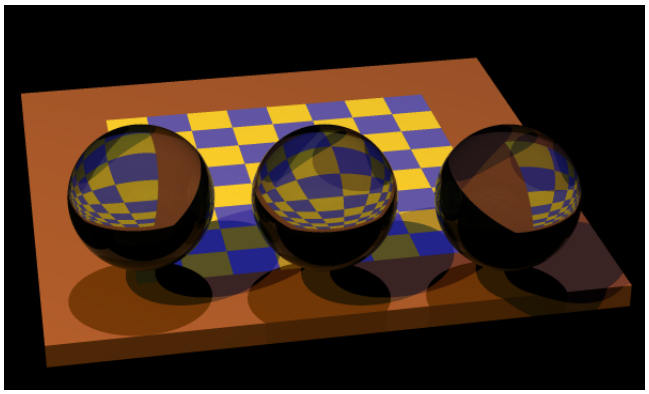

Figure 1.2: Another ray-traced image.

Chapter 2

Digital Images

2.1 Pixels and colors

An image is made of a rectangular grid of tiny square dots called *pixels*. Each pixel may take on a wide range of colors and brightness levels. In the C++ code that accompanies this tutorial, all images will be saved in PNG (Portable Network Graphics) files where each pixel has a red, green, and blue color component value. Each color component can range from 0 (meaning that color is absent, i.e. completely dark) to 255 (meaning that color is saturated or maximized). If a pixel has red, green, and blue values that are all 0, that pixel is black, but if all three values are 255, the pixel is white. Other combinations of red, green, and blue can reproduce any imaginable color. Because the red, green, and blue color components of any pixel are independent and can assume 256 distinct values, the pixel itself can have $256^3 = 16{,}777{,}216$ distinct color values.

2.2 An example

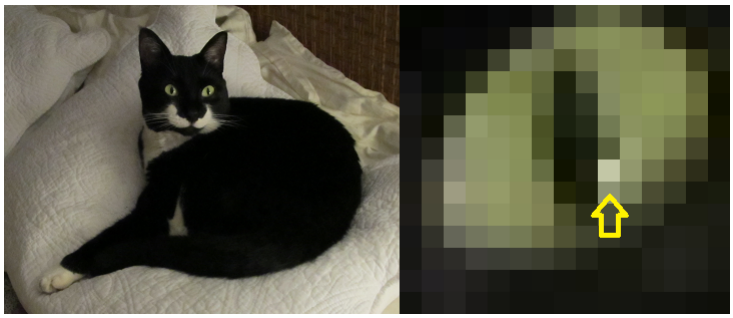

Figure 2.1: A digital image with cat's eye zoomed on right to show pixels.

In Figure 2.1 we see a photograph of my cat Lucci on the left, and a magnified view of one of her eyes on the right. Magnifying an ordinary picture like this reveals a mosaic of colored tiles, each tile being a separate pixel. The arrow points to a pixel whose color components are red=141, green=147, and blue=115. It is a remarkable fact of this digital age that photographs can be reduced to a long series of numbers in a computer.

2.3 Resolution

The amount of detail in a digital image, called *resolution*, increases with the number of pixels the image contains. Typical digital images have hundreds or even thousands of pixels on a side. For example, the cat image on the left side of Figure 2.1 is 395 pixels wide and 308 pixels high. Some images may have a much smaller pixel count if they are used for desktop icons, mouse pointers, or image thumbnails. Beyond a few thousand pixels on a side, images become unwieldy in terms of memory usage and bandwidth, so width and height dimensions of between 200 to 2000 pixels are the most common compromise of resolution and efficiency.

Chapter 3

Cameras, Eyes, and Ray Tracing

3.1 Images distort reality

Ray tracing is a technique that was inspired by the functioning of the human eye and devices like film cameras and TV cameras. Eyes and cameras receive light from a three-dimensional environment and focus it onto a two-dimensional surface. Whether this surface is flat (as in a film camera or a digital camera) or curved (as the eye's retina), the focused image has lost a dimension compared to the reality it represents. Just like no paper map can represent the spherical Earth without distortion, a focused image in a camera or eye causes distortions. Such distortions are so familiar to us in our everyday lives, and our brains so adept at processing the images to extract useful information from them, that it takes deliberate thought to even notice these effects as distortion:

- The image of a distant object is smaller than the image of a nearer object of the same physical size.
- Parallel lines that recede from the viewer appear to converge at a point at infinity, an effect called *perspective* in art. Such lines on an image actually do converge toward a point, even though in reality the lines are the same distance apart all along their lengths.
- An object's image will assume different shapes depending on the angle from which it is viewed. For example, Figure 3.1 shows how a coin can appear as a circular disc if viewed perpendicularly to its face, as an oval shape at a 45° angle, or even as a thin rectangular strip if viewed edge-on.

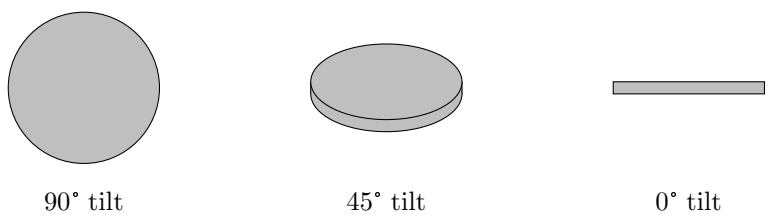

Figure 3.1: A coin viewed at different angles.

- The image of an opaque object includes only portions of its exterior surface that more or less face toward the viewer. The object's opaque matter blocks light reflected from its far surface from reaching the viewer.
- An object with uniform color (such as a piece of chalk) will appear to have light and dark regions based on the angle of light shining on it. Also, other objects may cast shadows on an object by blocking light from hitting some parts of its surface.

Again, these effects on the image are real, not imaginary; to accurately portray a 3D scene, they must be incorporated into the construction of any 2D image for a person to perceive that image as realistic.

3.2 Perspective in art

Any trained artist will appreciate the technical skill required to depict a three-dimensional scene convincingly on a flat canvas. Mastering the subtle use of light and shadow and the precise geometry of perspective require a great deal of practice and refinement.

But we cannot expect a computer to have a creative and artistic sense of expression. For a computer to create a realistic image, it must be told both what objects to draw and how to draw them. The three-dimensional objects must be modeled as mathematical formulas, and the optics of a camera must be simulated somehow in order to create a flat image from these formulas.

Contemplating this, one might despair after inspecting the innards of any modern camera. Whether of a more traditional film design or a more modern digital design, a high-end camera includes a complex arrangement of lenses, along with mechanical parts to move them relative to each other and the focal plane, where they work together to form images. Even the simplest practical camera has a single lens of an exquisitely precise shape, multiple layers of materials having different optical properties, and special chemical coatings to reduce internal reflections and glare.

Fortunately, we can avoid the conceptual difficulties of optical lenses in our computer simulations entirely. We can do this by searching back in history. Over 2000 years ago Aristotle (and others) recorded that a flat surface with a small hole, placed between a candle flame and a wall, will cause an inverted image of the flame to appear on the wall. Aristotle correctly surmised from this observation that light travels in straight lines.

3.3 Cameras

In the Renaissance, artists learned to draw landscapes and buildings with accurate perspective by sitting in a darkened room with a small hole in one wall. On a bright, sunny day, an upside-down image of the outdoor scenery would be projected on the wall opposite the hole. The artist could then trace outlines of the scene on paper or canvas, and simply turn it right-side-up when finished. In 1604 the famous astronomer Johannes Kepler used the Latin phrase for "darkened room" to describe this setup: *camera obscura*, hence the origin of the word *camera*.

The discovery of photochemicals in the 1800s, combined with a miniaturized camera obscura, led to the invention of pinhole cameras. Instead of an entire room, a small closed box with a pinhole in one side would form an image on a photographic plate situated on the opposite side of the box. (See Figure 3.2.)

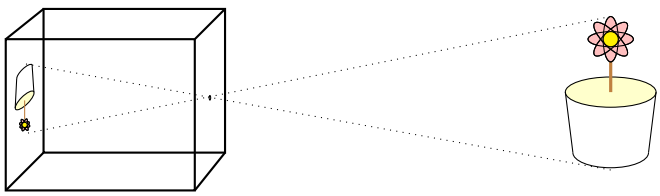

Figure 3.2: A pinhole camera.

The ray tracing algorithm presented here simulates an idealized pinhole camera. It models light traveling in straight lines passing through an extremely small hole (a perfect mathematical point, actually) and landing on a flat screen to form an image. However, for the sake of computational speed, ray tracing reverses the direction of the light rays. Instead of simulated rays of light reflecting from an object in all directions (as happens in reality), and only a tiny portion of them passing through the camera's pinhole, in ray tracing we start with each pixel on the imaginary screen, draw a straight line passing from that pixel through the pinhole and beyond to the scene outside the camera. This way, we ignore all rays of light that did not pass through the pinhole and are therefore irrelevant to the intended image. If the line strikes a point on the surface of some object in the scene, say a point on a bright red ball, we know that the associated pixel at the other end of the line should be colored red.

In a real pinhole camera, the light would have reflected from that spot on the red ball in all directions, but the only ray that would find its way onto the imaging screen would be the one that travels in the exact direction of the pinhole from that spot. Other spots on the ball (for

example, the parts facing away from the camera) would be blocked from the pinhole's view, and rays of light striking them would have no effect on the image at all. Reversing the direction of the light rays saves a tremendous amount of computing effort by eliminating consideration of all the irrelevant light rays.

Chapter 4

Vectors and Scalars

Before diving into the details of the ray tracing algorithm, the reader must have a good understanding of vectors and scalars. If you already feel comfortable with vectors, scalars, and mathematical operations involving them, such as a vector addition and subtraction, multiplying a vector with a scalar, and vector dot products and cross products, then you can skip this chapter and still understand the subsequent material. This chapter is not intended as an exhaustive treatment of vector concepts, but merely as a summary of the topics we need to understand ray tracing.

4.1 Scalars

A *scalar* is a number like 3.7 or -14.25. A scalar may be negative, positive, or zero. We use the word "scalar" to describe such numbers primarily so we can distinguish them from vectors.

4.2 Vectors

A *vector* is a triplet of numbers that can represent the location of a point in three-dimensional space or a direction in space from a given starting point. A vector is written as (x, y, z) where x, y, and z are real numbers called *components* or *coordinates*. The x, y, and z components indicate how far, and which way, to move along three perpendicular lines to arrive at a point. The perpendicular lines are called *axes* (the plural; the singular form is *axis*).

The starting point where the x, y, and z axes intersect is called the *origin*. At the origin, the values of x, y, and z are all zero, so the origin can be written as $(0, 0, 0)$. In Figure 4.1, starting at the origin $\mathbf{O}$, the vector $\mathbf{P} = (3, 2, 1)$ takes us 3 units to the right (or "east" if you prefer), 2 units toward the top of the page ("north") and 1 unit directly out from the page toward you ("up").

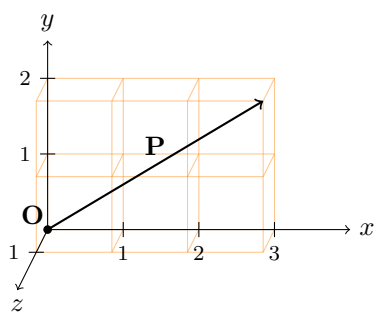

Figure 4.1: The vector $\mathbf{P} = (3, 2, 1)$. The origin is labeled $\mathbf{O}$.

Note that we can look at the x, y, and z axes from many different points of view. For example, we could let x point southeast and y point northeast. But not only do the x, y, and z axes have to be at right angles to each other, but the directions that any two of the axes point determine the direction the third axis must point. An easy way to visualize this requirement is to imagine

your head is at the origin and that you are looking in the direction of the x axis, with the top of your head pointing in the direction of the y axis. In this case the z axis must point to your right; it is not allowed to point to your left. This convention is arbitrary, but consistently following it is necessary in the mathematics and algorithms to come; otherwise certain vector operations will give wrong answers. (See also: Wikipedia's article *Right-hand rule.*)

4.3 Vector addition

Suppose that we start at the origin, but this time take a slightly more complicated journey.

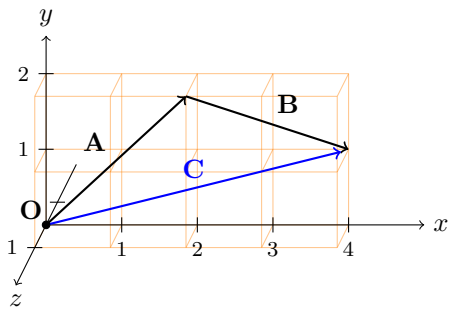

Figure 4.2: Vector addition of $\mathbf{A} = (2, 2, 1)$ and $\mathbf{B} = (2, -1, -1)$.

First we follow the vector $\mathbf{A}$ to move 2 units east, 2 units north, and 1 unit up. Then, starting from that point we move along the vector $\mathbf{B}$ and travel 2 more units east, one unit south (the same as -1 units north) and 1 unit down (-1 units up). We end up at a location specified by adding the respective x, y, and z components of the vectors $\mathbf{A}$ and $\mathbf{B}$, namely $(2 + 2, 2 + (-1), 1 + (-1)) = (4, 1, 0)$. (See Figure 4.2.) If we call our final position $\mathbf{C}$, we can write concisely in vector notation that

$$\mathbf{A} + \mathbf{B} = \mathbf{C}$$

or

$$(2, 2, 1) + (2, -1, -1) = (4, 1, 0).$$

It would not matter if we reversed the order of the vectors—if we went from the origin in the direction specified by the $\mathbf{B}$ vector first, then from there in the direction specified by the $\mathbf{A}$ vector, we would still end up at $\mathbf{C}$. So vectors added in either order, $\mathbf{A} + \mathbf{B}$ or $\mathbf{B} + \mathbf{A}$, always produce the same result, just like scalars can be added in either order to obtain the same scalar result. Said in more formal mathematical terms, vector addition and scalar addition are both *commutative*. In general, if

$$\mathbf{A} = (A_x, A_y, A_z)$$

and

$$\mathbf{B} = (B_x, B_y, B_z)$$

then

$$\mathbf{A} + \mathbf{B} = (A_x + B_x, A_y + B_y, A_z + B_z).$$

4.4 Vector subtraction

Similarly, we can define vector subtraction by

$$\mathbf{A} - \mathbf{B} = (A_x - B_x, A_y - B_y, A_z - B_z).$$

One helpful interpretation of vector subtraction is to imagine starting at the origin and moving along the vector $\mathbf{A}$ (as we did before for vector addition) but then moving in the opposite direction as specified by $\mathbf{B}$. For example, re-using $\mathbf{B} = (2, -1, -1)$, we would move 2 units west (instead of east), 1 unit north (instead of south), and 1 unit up (instead of down), to arrive at $(2 - 2, 2 - (-1), 1 - (-1)) = (0, 3, 2)$, labeled as $\mathbf{D}$ in Figure 4.3.

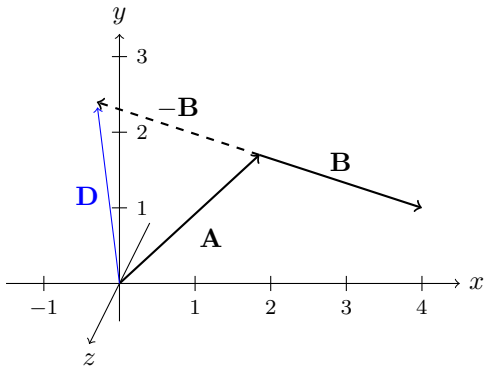

Figure 4.3: Vector subtraction: $\mathbf{D} = \mathbf{A} - \mathbf{B} = \mathbf{A} + (-\mathbf{B})$.

Another important interpretation of vector subtraction is that it calculates a vector that takes us from one location to another location. Suppose we are located at the point $\mathbf{E} = (1, 3, 0)$ and we want to know how to travel to the point $\mathbf{F} = (5, 2, 0)$. Another way to pose this question is to ask, what is the vector $\mathbf{G}$ such that $\mathbf{E} + \mathbf{G} = \mathbf{F}$? In algebra with scalars, we could subtract $\mathbf{E}$ from both sides of the equation and determine that $\mathbf{G} = \mathbf{F} - \mathbf{E}$. It turns out that this approach is just as valid when applied to vectors. So in our current example, $\mathbf{G} = \mathbf{F} - \mathbf{E} = (5, 2, 0) - (1, 3, 0) = (5 - 1, 2 - 3, 0 - 0) = (4, -1, 0)$. Looking at Figure 4.4, it makes sense that if we start at $\mathbf{E}$, move 4 units east, 1 unit south, and 0 units up, we end up at $\mathbf{F}$.

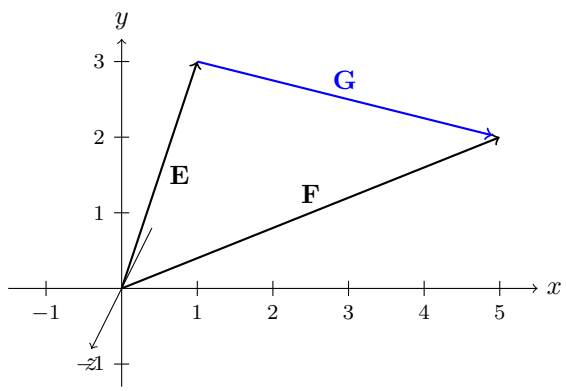

Figure 4.4: To travel from $\mathbf{E}$ to $\mathbf{F}$, move by $\mathbf{G} = \mathbf{F} - \mathbf{E}$.

4.5 Scalars multiplied by vectors

Suppose we want to start at the origin point $(0, 0, 0)$ and move in the direction of some vector $\mathbf{A}$, but move 3 times as far as $\mathbf{A}$ goes? Based on what we have seen so far, we could calculate $\mathbf{A} + \mathbf{A} + \mathbf{A}$. If we were dealing with a scalar quantity, we would immediately be able to simplify this as $\mathbf{A} + \mathbf{A} + \mathbf{A} = 3\mathbf{A}$. Does this work for vectors? Let's find out.

$$\begin{aligned}
&\mathbf{A} + \mathbf{A} + \mathbf{A} \\
=&(A_x, A_y, A_z) + (A_x, A_y, A_z) + (A_x, A_y, A_z) \\
=&(A_x + A_x + A_x, A_y + A_y + A_y, A_z + A_z + A_z) \\
=&(3A_x, 3A_y, 3A_z)
\end{aligned}$$

If we allow ourselves to write the final vector as $(3A_x, 3A_y, 3A_z) = 3\mathbf{A}$, then we have behavior for vectors that matches the familiar rules of scalar algebra. In general, we define multiplication of a scalar u and a vector $\mathbf{A}$ as

$$u\mathbf{A} = u(A_x, A_y, A_z) = (uA_x, uA_y, uA_z)$$

where u is any scalar value, whether negative, positive, or zero. If $u = 0$, the resulting product $u\mathbf{A} = (0, 0, 0)$, no matter what the values of A_x, A_y, and A_z are. Another way to think of this is that moving zero times in any direction is the same as not moving at all. Multiplying $u = -1$ by a vector is the same as moving the same distance as that vector, but in the opposite direction. We can write $(-1)\mathbf{A}$ more concisely as $-\mathbf{A}$.

4.6 Vector magnitude

Sometimes we will have a vector and need to know how long it is. The length of a vector is called its *magnitude*. Mathematicians express the magnitude of a vector by writing it between two vertical bars. So the magnitude of a vector $\mathbf{A}$ is written like this: $|\mathbf{A}|$. $|\mathbf{A}|$ is a scalar, because it is just a simple number. If $\mathbf{A} = (A_x, A_y, A_z)$, then $|\mathbf{A}|$ can be calculated using the Pythagorean Theorem as

$$|\mathbf{A}| = \sqrt{A_x^2 + A_y^2 + A_z^2}$$

For example, if $\mathbf{A} = (3, -1, 2)$, then

$$|\mathbf{A}| = \sqrt{3^2 + (-1)^2 + 2^2} = \sqrt{14} = 3.7416...$$

4.7 Unit vectors

Sometimes it will be handy to use a vector to specify a direction in space while requiring the vector's magnitude to be exactly equal to 1. We call any vector whose magnitude is 1 a *unit vector*. If we have an arbitrary vector $\mathbf{A} = (A_x, A_y, A_z)$ and want to find a unit vector $\hat{\mathbf{v}}$ that points in the same direction, we can do this by dividing the vector $\mathbf{A}$ by its scalar length $|\mathbf{A}|$:

$$\hat{\mathbf{v}} = \frac{\mathbf{A}}{|\mathbf{A}|} = \left(\frac{A_x}{|\mathbf{A}|}, \frac{A_y}{|\mathbf{A}|}, \frac{A_z}{|\mathbf{A}|}\right)$$

where again, $|\mathbf{A}| = \sqrt{A_x^2 + A_y^2 + A_z^2}$. (Throughout this text, I will use a lowercase letter with a caret over it to represent unit vectors, so if you see a variable like $\hat{\mathbf{v}}$, you can assume that $|\hat{\mathbf{v}}| = 1$.) The only case where the above magnitude formula fails is when $\mathbf{A} = (0, 0, 0)$, because we would end up trying to divide by 0. It makes sense that $(0, 0, 0)$ does not "point" in any particular direction, so it is the one case where it becomes meaningless to speak of an associated unit vector.

4.8 Vector dot products

A useful question we can ask about two different vectors $\mathbf{A}$ and $\mathbf{B}$ is, do $\mathbf{A}$ and $\mathbf{B}$ point in the same direction, opposite directions, are they perpendicular to each other, or are they somewhere between these special cases? There is a surprisingly easy way to answer this question, though explaining why it works is beyond the scope of this book. By multiplying each of the three components of $\mathbf{A}$ with the corresponding component of $\mathbf{B}$, and adding up the three resulting products, we obtain a sum called the *dot product* of the vectors $\mathbf{A}$ and $\mathbf{B}$, so called because mathematicians write a dot "$\cdot$" between them:

$$\mathbf{A} \cdot \mathbf{B} = (A_x, A_y, A_z) \cdot (B_x, B_y, B_z) = A_x B_x + A_y B_y + A_z B_z$$

Note that although $\mathbf{A}$ and $\mathbf{B}$ are both vectors, their dot product is a scalar. For this reason, some authors will refer to this operation as the *scalar product* of the two vectors. It turns out that if $\mathbf{A}$ and $\mathbf{B}$ are perpendicular to each other, their dot product $\mathbf{A} \cdot \mathbf{B}$ will be 0. As a simple example, consider the vectors $(3, 2, 0)$ and $(2, -3, 0)$. Their dot product is $(3, 2, 0) \cdot (2, -3, 0) = (3)(2) + (2)(-3) + (0)(0) = 0$. If the angle between the two vectors is less than 90°, their dot product is a positive number. If the angle is greater than 90°, the dot product is negative. Thus dot products are helpful for determining whether two vectors are perpendicular, roughly pointing in the same direction, or roughly pointing in opposite directions.

Dot products also depend on the magnitudes of the two vectors: assuming that the dot product is not zero (the case when the vectors are not perpendicular), the larger the magnitude of **A** or **B** is, the larger the dot product becomes. All of these facts are summarized by the following equation, which relates the angle between any two vectors and their respective magnitudes.

$$\mathbf{A} \cdot \mathbf{B} = |\mathbf{A}||\mathbf{B}| \cos(\theta)$$

where θ is the angle between **A** and **B**. If you are rusty on trigonometry, the $\cos(\theta)$ part is read "cosine of theta." For now, all you need to know is that:

- $\cos(0^\circ) = 1$
- $\cos(\theta)$ is between 0 and 1 when θ is between 0° and 90°
- $\cos(90^\circ) = 0$
- $\cos(\theta)$ is between 0 and -1 when θ is between 90° and 180°
- $\cos(180^\circ) = -1$

I will explain the cosine and sine trigonometry functions in more detail later.

4.9 Vector cross products

Another important question we will need to answer about two vectors is, what directions in space are perpendicular to both vectors? This question does not make sense in all cases. If either of the vectors has zero magnitude (i.e., is (0, 0, 0)), there is no meaningful notion of another vector being parallel or perpendicular to it. Also, if the angle between two vectors is 0° (they both point in exactly the same direction) or 180° (they point in exactly opposite directions, as shown in Figure 4.5), there are an infinite number of directions perpendicular to both.

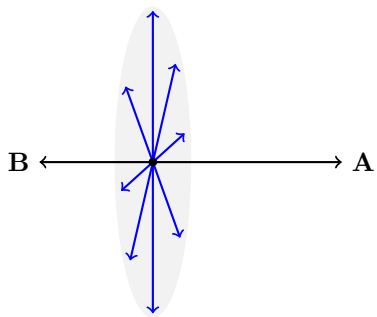

Figure 4.5: An infinite number of vectors are perpendicular to a pair of vectors **A** and **B** that point in opposite directions.

But if both vectors have positive magnitudes and the angle between them is somewhere between 0° and 180°, then there is a unique plane that passes through both vectors. In order for a third vector to point in a direction perpendicular to both **A** and **B**, that vector must be perpendicular to this plane. In fact, there are only two such directions, and they lie on opposite sides of the plane. If the vector **C** points in one of these perpendicular directions, then $-\mathbf{C}$ points in the other, as shown in Figure 4.6.

Finding a vector like **C** would answer our original question, regardless of the magnitude of **C**: it would point in a direction at right angles to both **A** and **B**.

There is a formula for calculating a perpendicular vector **C** from any two vectors **A** and **B**, though it is more complicated than the dot product formula. It is called a *cross product*, named after the "×" in the conventional math notation "$\mathbf{A} \times \mathbf{B}$". Without derivation in this book, here is the formula for the cross product:

$$\begin{aligned}\mathbf{A} \times \mathbf{B} &= (A_x,\, A_y,\, A_z) \times (B_x,\, B_y,\, B_z) \\ &= (A_y B_z - A_z B_y,\, A_z B_x - A_x B_z,\, A_x B_y - A_y B_x)\end{aligned}$$

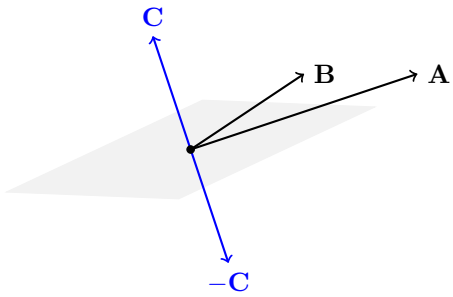

Figure 4.6: There are two directions $\mathbf{C}$ and $-\mathbf{C}$ perpendicular to the plane of two non-parallel vectors $\mathbf{A}$ and $\mathbf{B}$.

If we let $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, then $\mathbf{C}$ points in the direction as shown in Figure 4.6. The cross product taken in the other order, $\mathbf{B} \times \mathbf{A}$, points in the opposite direction, and is in fact equal to $-\mathbf{C}$. To visualize the direction of $\mathbf{C} = \mathbf{A} \times \mathbf{B}$ for any two vectors $\mathbf{A}$ and $\mathbf{B}$, imagine yourself oriented in space such that $\mathbf{A}$ is on your right and $\mathbf{B}$ is on your left. (This might require your body to be sideways or even upside-down, but fortunately this is only in your imagination!) Then $\mathbf{C} = \mathbf{A} \times \mathbf{B}$ will point the same way as the top of your head. Cross products share one trait with dot products: they get "bigger" as the magnitude of $\mathbf{A}$ or $\mathbf{B}$ increases. But a cross product is a vector, not a scalar like the dot product. Another difference is that the dot product of two perpendicular vectors is zero, but the cross product of two perpendicular vectors maximizes the magnitude of their cross product. If both $|\mathbf{A}| > 0$ and $|\mathbf{B}| > 0$, the only way to get a cross product with a zero magnitude is when $\mathbf{A}$ and $\mathbf{B}$ are pointing in exactly the same (0°) or exactly opposite (180°) directions. The general formula for a cross product's magnitude is

$$|\mathbf{A} \times \mathbf{B}| = |\mathbf{A}||\mathbf{B}| \sin(\theta)$$

Compare this to the formula relating the dot product and the included angle θ. Just as $\cos(\theta)$ is zero when $\theta = 90°$, $\sin(\theta)$ is zero when $\theta = 0°$ or $180°$.

4.10 Sine and cosine: a crash course

If you are new to trigonometry, or you once knew it but have forgotten everything, this section will help. There is a lot to the subject of trigonometry, but I will cover only the essentials you will need for understanding the ray tracing topics covered in this book. Specifically, we will explore the sine and cosine functions and learn a simple way to visualize what they mean.

Let's start with a circle on the xy plane whose radius is 1 unit and centered at the origin $(0, 0)$, as shown in Figure 4.7.

If we draw a line at an angle θ measured counterclockwise from the $+x$ axis and find the point $\mathbf{P}$ where that line crosses the circle, then $\cos(\theta)$ is the point's x coordinate (how far $\mathbf{P}$ is to the right or left of the origin) and $\sin(\theta)$ is the point's y coordinate (how far $\mathbf{P}$ is above/below the origin). If $\mathbf{P} = (P_x, P_y)$, then $P_x = \cos(\theta)$, read "P sub x equals cosine of theta," and $P_y = \sin(\theta)$, read "P sub y equals sine of theta." So if you want to know, for example, when $\cos(\theta)$ is zero, ask yourself what angles θ could be to make $\mathbf{P}$ neither to the left nor right of the origin. If we are talking about θ being the angle between two vectors, θ must be in the range 0°...180°, and therefore for P_x to be zero, $\mathbf{P}$ must be at $(0, +1)$, forcing θ to be 90°.

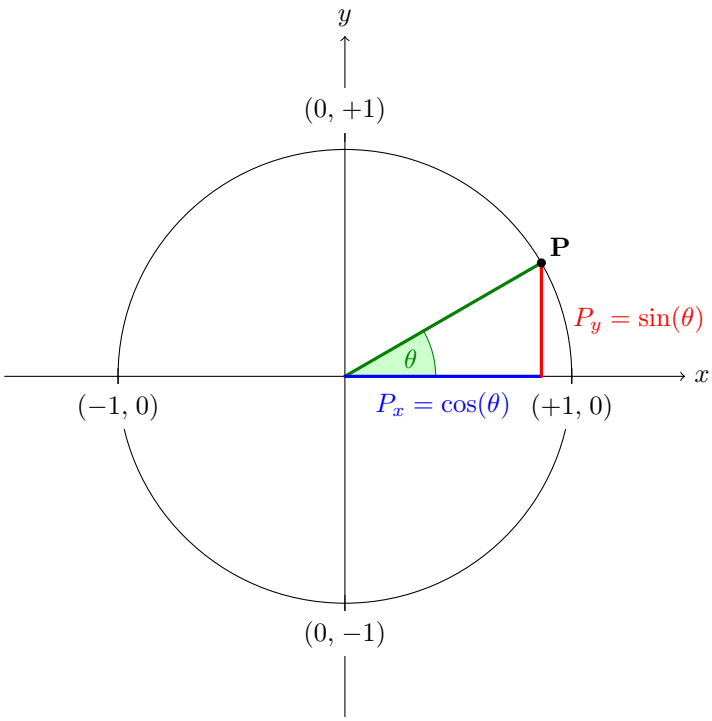

Figure 4.7: Sine and cosine of an angle θ

Chapter 5

C++ Code for Ray Tracing

5.1 Platforms and redistribution

This book is accompanied by C++ code that you can download for free and build as-is or modify. The code will build and run without modification on Windows, Linux, and Mac OS X. I built it on Windows 7 using Microsoft Visual Studio 2008, and on both Ubuntu Linux 9.10 and Mac OS X 10.8.2 using a simple script that launches `g++`, the C++ front end for the GNU compiler suite (or LLVM on OS X). All of my code is open source. You may use and re-distribute this code (or portions of it) for your own private, educational, or commercial purposes, so long as the comment blocks at the front of each source file with my name, web site address, and copyright and legal notices are left intact. See the legal notices at the front of any of the source files (.cpp or .h) for more details.

Please note that I am **not** the author of the files `lodepng.h` and `lodepng.cpp` for writing PNG formatted image files; these were created by Lode Vandevenne. For more information about LodePNG, please visit `http://lodev.org/lodepng`. Both LodePNG and my ray tracing code are provided with similar open source licenses.

5.2 Downloading and installing

Download the zip file

```
http://cosinekitty.com/raytrace/rtsource.zip
```

to a convenient location on your hard drive. The easiest way is to use your web browser and let it prompt you for the location where you want to save the zip file. Alternatively, in Linux or OS X you can go into a terminal window (where you will need to end up soon anyway) and enter the following command from somewhere under your home folder:

```
curl http://cosinekitty.com/raytrace/rtsource.zip -o rtsource.zip
```

After downloading the file, unzip its contents. If there is an option in your unzip program to preserve the subdirectory structure encoded in the zip file, be sure to enable it. Linux or OS X users can use the `unzip` command:

```
unzip rtsource.zip
```

This will create a new subdirectory called `raytrace` beneath your current directory that contains everything you need to follow along with the rest of this book.

5.3 Building on Windows

Use Microsoft Visual C++ to open the solution file

```
...\raytrace\raytrace.sln
```

where ... represents the path where you unzipped `rtsource.zip`. Then choose the menu

item Build, followed by Batch Build to compile all the code. Check the boxes for both Debug and Release configurations, then click the Build button. The code should compile and link with no warnings and no errors.

5.4 Getting ready to build under OS X

If you do any software development for the Macintosh, you have probably already installed some version of Apple's integrated development environment, Xcode. However, your machine may or may not be ready to build C++ programs from the command line. To find out, open a Terminal window and type the command `g++`. If you get a "command not found" error, you will need to follow a couple of steps to get the command-line build tools installed. Here is what I did using Xcode 4.6:

1. Open up Xcode.
2. Under the Xcode menu at the top of your screen, choose Preferences.
3. A box will appear. Choose the Downloads tab.
4. Look for a "Command Line Tools" entry. Click the Install button to the right of it.
5. Wait for the command line tools to download and install.
6. Go back into your Terminal window. When you enter the `g++` command again, it should no longer say "command not found" but should instead say something like `i686-apple-darwin11-llvm-g++-4.2: no input files`.

If you do not have Xcode installed, and you don't want to download 1.65 GB just to build from the command line, here is a more palatable alternative. You will need an Apple ID, but you can create one for free. Once you have an Apple ID, take a look at the Downloads for Apple Developers page whose URL is listed below and find the Command Line Tools package appropriate for your version of OS X. The download will be much smaller (about 115 MB or so) and will cost you nothing.

```
https://developer.apple.com/downloads/index.action
```

Once you get `g++` installed, you are ready to proceed to the next section.

5.5 Building on Linux and OS X

Go into a command prompt (terminal window) and change to the directory where you unzipped `rtsource.zip`. Then you will need to go into the two levels deep of directories both named `raytrace`:

```
cd raytrace/raytrace
```

The first time before you build the code for Linux, you will need to grant permission to run the build script as an executable:

```
chmod +x build
```

And finally, build the code:

```
./build
```

You should see the computer appear to do nothing for a few seconds, then return to the command prompt. A successful build will not print any warnings or errors, and will result in the executable file `raytrace` in the current directory: if you enter the command

```
ls -l raytrace
```

you should see something like

```
-rwxr-xr-x 1 dcross dcross 250318 2012-10-29 15:17 raytrace
```

5.6 Running the unit tests

Once you get the C++ code built on Windows, Linux, or OS X, running it is very similar on any of these operating system. On Windows the executable will be

```
raytrace\Release\raytrace.exe
```

On Linux or OS X, it will be

```
raytrace/raytrace/raytrace
```

In either case, you should run unit tests to generate some sample PNG images to validate the built code and see what kinds of images it is capable of creating. The images generated by the unit tests will also be an important reference source for many parts of this book.

Go into a command prompt (terminal window) and change to the appropriate directory for your platform's executable, as explained above. Then run the program with the `test` option. (You can always run the `raytrace` program with no command line options to display help text that explains what options are available.) On Linux or OS X you would enter

```
./raytrace test
```

And on Windows, you would enter

```
raytrace test
```

You will see a series of messages that look like this:

```
Wrote donutbite.png
```

When finished, the program will return you to the command prompt. Then use your web browser or image viewer software to take a peek at the generated PNG files. On my Windows 7 laptop, here is the URL I entered in my web browser to see all the output files.

```
file:///C:/don/dev/trunk/math/block/raytrace/raytrace/
```

On my OS X machine, the URL looks like this:

```
file:///Users/don/raytrace/raytrace/
```

Yours will not be exactly the same as these, but hopefully this gets the idea across. If everything went well, you are now looking at a variety of image files that have `.png` extensions. These image files demonstrate many of the capabilities of the `raytrace` program. Take a few moments to look through all these images to get a preview of the mathematical and programming topics to come.

5.7 Example of creating a ray-traced image

Let's dive into the code and look at how the double-torus image in Figure 1.1 was created. Open up the source file `main.cpp` with your favorite text editor, or use Visual Studio's built-in programming editor. In Visual Studio, you can just double-click on `main.cpp` in the Solution Explorer; `main.cpp` is listed under Source Files. Using any other editor, you will find `main.cpp` in the directory `raytrace/raytrace`. Once you have opened `main.cpp`, search for the function `TorusTest`. It looks like this:

```
void TorusTest(const char* filename, double glossFactor)
{
    using namespace Imager;

    Scene scene(Color(0.0, 0.0, 0.0));

    const Color white(1.0, 1.0, 1.0);
    Torus* torus1 = new Torus(3.0, 1.0);
    torus1->SetMatteGlossBalance(glossFactor, white, white);
    torus1->Move(+1.5, 0.0, 0.0);

    Torus* torus2 = new Torus(3.0, 1.0);
    torus2->SetMatteGlossBalance(glossFactor, white, white);
    torus2->Move(-1.5, 0.0, 0.0);
    torus2->RotateX(-90.0);

    SetUnion *doubleTorus = new SetUnion(
        Vector(0.0, 0.0, 0.0),
        torus1,
        torus2
    );

    scene.AddSolidObject(doubleTorus);

    doubleTorus->Move(0.0, 0.0, -50.0);
    doubleTorus->RotateX(-45.0);
    doubleTorus->RotateY(-30.0);

    // Add a light source to illuminate the objects
    // in the scene; otherwise we won't see anything!
    scene.AddLightSource(
        LightSource(
            Vector(-45.0, +10.0, +50.0),
            Color(1.0, 1.0, 0.3, 1.0)
        )
    );

    scene.AddLightSource(
        LightSource(
            Vector(+5.0, +90.0, -40.0),
            Color(0.5, 0.5, 1.5, 0.5)
        )
    );

    // Generate a PNG file that displays the scene...
    scene.SaveImage(filename, 420, 300, 5.5, 2);
    std::cout << "Wrote " << filename << std::endl;
}
```

Don't worry too much about any details you don't understand yet — everything will be explained in great detail later in this book. For now, just notice these key features:

1. We create a local variable of type `Scene`. We specify that its background color is `Color(0.0, 0.0, 0.0)`, which is black.

2. We create a `Torus` object called `torus1` and move it to the right (in the $+x$ direction) by 1.5 units.

3. We create another `Torus` called `torus2`, move it left by 1.5 units, and rotate it 90° clockwise around its x-axis (technically, about an axis parallel to the x-axis that passes through the object's center, but more about that later). Rotations using negative angle values, as in this case, are clockwise as seen from the positive axis direction; positive angle values indicate counterclockwise rotations.

4. We create a `SetUnion` of `torus1` and `torus2`. For now, just think of `SetUnion` as a container that allows us to treat multiple visible objects as a single entity. This variable is called `doubleTorus`.

5. We call `scene.AddSolidObject()` to add `doubleTorus` to the scene. The scene is responsible for deleting the object pointed to by `doubleTorus` when `scene` gets destructed. The

destructor `Scene::~Scene()` will do this automatically when the code returns from the function `TorusTest`. That destructor is located in `imager.h`.

6. The `doubleTorus` is moved 50 units in the $-z$ direction, away from the camera and into its prime viewing area. (More about the camera later.) Then we rotate `doubleTorus`, first 45° around its center's x-axis, then 30° around its center's y-axis. Both rotations are negative (clockwise). Note that we can move and rotate objects either before or after adding them to the `Scene`; order does not matter.

7. We add two point light sources to shine on the double torus, one bright yellow and located at $(-45, 10, 50)$, the other a dimmer blue located at $(5, 90, -40)$. As noted in the source code's comments, if we forget to add at least one light source, nothing but a solid black image will be generated (not very interesting).

8. Finally, now that the scene is completely ready, with all visible objects and light sources positioned as intended, we call `scene.SaveImage()` to ray-trace an image and save it to a PNG file called `torus.png`. This is where almost all of the run time is: doing all the ray tracing math.

5.8 The `Vector` class

The `Vector` class represents a vector in three-dimensional space. We use double-precision floating point for most of the math calculations in the ray tracing code, and the members `x`, `y`, and `z` of class `Vector` are important examples of this. In addition to the raw component values stored as member variables, `class Vector` has two constructors, some member functions, and some helper inline functions (including overloaded operators). The default constructor initializes a `Vector` to $(0, 0, 0)$. The other constructor initializes a `Vector` to whatever x, y, z values are passed into it.

The member function `MagnitudeSquared` returns the square of a vector's length, or $x^2 + y^2 + z^2$, and `Magnitude` returns the length itself, or $\sqrt{x^2 + y^2 + z^2}$. `MagnitudeSquared` is faster when code just needs to compare two vectors to see which is longer/shorter, because it avoids the square root operation as needed in `Magnitude`.

The `UnitVector` member function returns a vector that points in the same direction as the vector it is called on, but whose length is one. `UnitVector` does not modify the value of the original `Vector` object that it was called on; it returns a brand new, independent instance of `Vector` that represents a unit vector.

Just after the declaration of `class Vector` are a few inline functions and overloaded operators that greatly simplify a lot of the vector calculation code in the ray tracer. We can add two vector variables by putting a "`+`" between them, just like we could with numeric variables:

```
Vector a(1.0, 5.0, -3.2);
Vector b(2.0, 3.0, 1.0);
Vector c = a + b;    // c = (3.0, 8.0, -2.2)
```

Likewise, the "`-`" operator is overloaded for subtraction, and "`*`" allows us to multiply a scalar and a vector. The inline function `DotProduct` accepts two vectors as parameters and returns the scalar dot product of those vectors. `CrossProduct` returns the vector-valued cross product of two vectors.

5.9 `struct LightSource`

This simple data type represents a point light source. It emits light of a certain color (specified by red, green, and blue values) in all directions from a given point (specified by a position vector). At least one light source must be added to a `Scene`, or nothing but a black silhouette will be visible against the `Scene`'s background color. (If that background color is also black, the rendered image will be completely black.) Often, a more pleasing image results from using multiple light sources with different colors positioned so that they illuminate objects from different angles. It is also possible to position many `LightSource` instances near each other to produce an approximation of blurred shadows, to counteract the very sharp shadows created by a single `LightSource`.

5.10 The `SolidObject` class

The abstract class `SolidObject` is the base class for all the three-dimensional bodies that this ray-tracing code draws. It defines methods for rotating and moving solid objects to any desired orientation and position in a three-dimensional space. It also specifies a contract by which calling code can determine the shape of its exterior surfaces, a method for determining whether a body contains a given point in space, and methods for determining the optical properties of the body.

All `SolidObject` instances have a center point, about which the object rotates via calls to `RotateX`, `RotateY`, and `RotateZ`. These methods rotate the entire object about the center point by the specified number of degrees counterclockwise as seen looking at that center point from the positive direction along the specified axis. For example, `RotateX(15.0)` will rotate the object 15° counterclockwise around a line parallel to the x-axis that passes through the object's center point, as seen from the $+x$ direction. To rotate 15° clockwise instead, you can do `Rotate(-15.0)`.

The `Translate` method moves the object by the specified Δx, Δy, and Δz amounts. The `Move` methods are similar, but they move an object such that its center point lands at the specified location. In other words, the parameters to `Translate` are changes in position, but the parameters to `Move` are absolute coordinates.

5.11 `SolidObject::AppendAllIntersections`

This member function is of key importance to the ray tracing algorithm. It is a pure virtual method, meaning it must be implemented differently by any derived class that creates a visible shape in a rendered image. The caller of `AppendAllIntersections` passes in two vectors that define a ray: a *vantage point* (a position vector that specifies a location in space from which the ray is drawn) and a direction vector that indicates which way the ray is aimed. `AppendAllIntersections` generates zero or more `Intersection` structs, each of which describes the locations (if any) where the ray intersects with the opaque exterior surfaces of the solid object. The new intersections are appended to the list parameter `intersectionList`, which may already contain other intersections before `AppendAllIntersections` is called.

As an example, the class `Sphere`, which derives from `SolidObject`, contains a method `Sphere::AppendAllIntersections` that may determine that a particular direction from a given vantage point passes through the front of the `Sphere` at one point and emerges from the back of the `Sphere` at another point. In this case, it inserts two extra `Intersection` structs at the back of `intersectionList`.

5.12 `struct Intersection`

The `Intersection` struct is a "plain old data" (POD) type for runtime efficiency. This means that any code that creates an `Intersection` object must take care to initialize all its members. Any members left uninitialized will have invalid default values. Here are the members of `Intersection`:

- `distanceSquared` : The square of the distance from the vantage point to the intersection point, as traveled along the given direction. As mentioned earlier, squared distances work just fine for comparing two or more distances against each other, without the need for calculating a square root.

- `point` : The position vector that represents the point in space where an intersection with the surface of an object was found.

- `surfaceNormal` : A unit vector (again, a vector whose length is one unit) that points outward from the interior of the object, and in the direction perpendicular to the object's surface at the intersection point.

- `solid` : A pointer to the solid object that the ray intersected with. Depending on situations which are not known ahead of time, this object may or may not need to be interrogated later for more details about its optical properties at the intersection point or some other point near the intersection.

- `context` : Most of the time, this field is not used, but some classes (e.g., `class TriangleMesh`, which is discussed later) need a way to hold extra information about the intersection point to help them figure out more about the optical properties there later. For the majority of cases where `context` is not needed, it is left as a null pointer.

Figure 5.1 shows a vantage point $\mathbf{D}$ and a direction vector $\mathbf{E}$ which points along an infinite ray that intersects a sphere in two places, $\mathbf{P_1}$ and $\mathbf{P_2}$. The surface normal unit vectors $\hat{\mathbf{n}}_\mathbf{1}$ and $\hat{\mathbf{n}}_\mathbf{2}$ each point outward from the sphere and in a direction perpendicular to the sphere's surface at the respective intersection points $\mathbf{P_1}$ and $\mathbf{P_2}$.

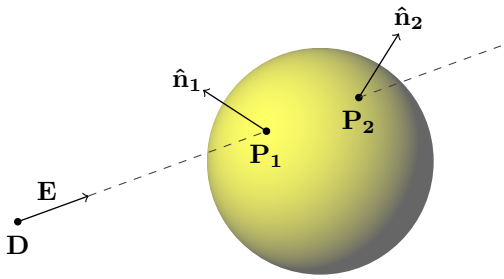

Figure 5.1: A direction vector $\mathbf{E}$ from a vantage point $\mathbf{D}$ tracing along an infinite ray. The ray intersects the sphere in two places, $\mathbf{P_1}$ and $\mathbf{P_2}$, with $\hat{\mathbf{n}}_\mathbf{1}$ and $\hat{\mathbf{n}}_\mathbf{2}$ as their respective surface normal unit vectors.

5.13 `Scene::SaveImage`

We encountered this method earlier when we looked at the function `TorusTest` in `main.cpp`. Now we are ready to explore in detail how it works. `Scene::SaveImage` is passed an output PNG filename and integer width and height values that specify the dimensions of a rectangular image to create, both expressed in pixels. The method simulates a camera point located at the origin — the point $(0, 0, 0)$ — and maps every pixel to a point on an imaginary screen that is offset from that camera point. The camera is aimed in the $-z$ direction (**not** the $+z$ direction) with the $+x$ axis pointing to the right and the $+y$ axis pointing upward. This way of aiming the camera allows the three axes to obey the right-hand rule while preserving the familiar orientation of $+x$ going to the right and $+y$ going up. Any visible surfaces in a scene therefore must have negative z values.

Our simulated camera works just like a pinhole camera, only we want it to create a right-side-up image, not an image that is backwards and upside-down. One way to accomplish this is to imagine that the pinhole camera's screen is translucent (perhaps made of wax paper) and that we are looking at it from behind while hanging upside-down.

There is an equivalent way to conceive of this apparatus so as to obtain the same image, without the awkwardness of hanging upside-down: instead of a translucent screen behind the camera point, imagine a completely invisible screen in front of the camera point. We can still break this screen up into pixels, only this time we imagine rays that start at the camera point and pass through each pixel on the screen. Each ray continues through the invisible screen and out into the space beyond, where it may or may not strike the surface of some object. Because all objects in this ray tracer are completely opaque, whichever point of intersection (if any) is closest to the camera will determine the color of the screen pixel that the ray passed through.

It doesn't matter which of these two models you imagine; the formulas for breaking up the screen into pixels will be the same either way, and so will the resulting C++ code. If you open up the source file `scene.cpp` and find the code for `Scene::SaveImage`, you will see that it requires two other arguments in addition to the filename, pixel width, and pixel height: one called `zoom` and the other called `antiAliasFactor`. The `zoom` parameter is a positive number that works like the zoom lens on a camera. If `zoom` has a larger numeric value, `SaveImage` will create an image that includes a wider and taller expanse of the scenery — the objects will look smaller but more will fit inside the screen. Passing a smaller value for `zoom` will cause objects to appear larger by including a smaller portion of the scene.

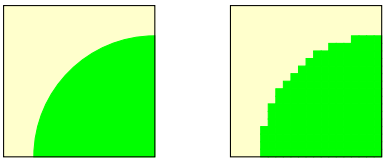

Figure 5.2: An idealized image on the left, and aliasing of it on the right when converted to a 20×20 grid of pixels.

The final parameter, `antiAliasFactor`, is an integer that allows the caller to adjust the compromise between faster rendering speed and image quality. We need to pause here to consider a limitation of ray tracing. Because there are only a finite number of pixels in any digital image, and ray-traced images follow the path of a single ray for each pixel on the screen, it is possible to end up with unpleasant distortion in the image known as *aliasing*. Figure 5.2 shows an example of aliasing. On the left is an image of a quarter-circle. On the right is how the image would appear when rendered using a grid of 20 pixels wide by 20 pixels high.

The jagged boundary, sometimes likened to stair steps, is what we mean by the word *aliasing*. It results from the all-or-nothing approach of forcing each pixel to take on the color of a single object. The `antiAliasFactor` provides a way to reduce (but not eliminate) aliasing by passing multiple rays at slightly different angles through each pixel and averaging the color obtained for these rays into a single color that is assigned to the pixel.

For example, passing `antiAliasingFactor = 3` will break each pixel into a 3×3 grid, or 9 rays per pixel, as shown in Figure 5.3.

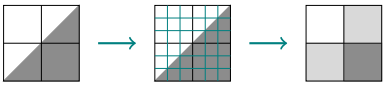

Figure 5.3: Anti-aliasing of a 2×2 image by breaking each pixel into a 3×3 grid and averaging the colors.

Figure 5.4 shows a comparison of the same scene imaged with `antiAliasFactor` set to 1 (i.e., no anti-aliasing) on the left, and 5 on the right.

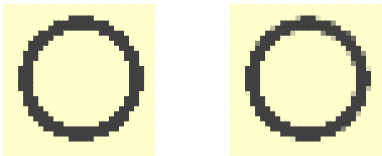

Figure 5.4: A scene without anti-aliasing (left) and with anti-aliasing (right).

There is a price to pay for using anti-aliasing: it makes the images take much longer to generate. In fact, the execution time increases as the square of the `antiAliasFactor`. You will find that setting `antiAliasFactor` to 3 will cause the image rendering to take 9 times as long, setting it to 4 will take 16 times as long, etc. There is also diminishing return in image quality for values much larger than 4 or 5. When starting the code to create a new image, I recommend starting with `antiAliasFactor = 1` (no anti-aliasing) until you get the image framing and composition just the way you like. Then you can increase `antiAliasFactor` to 3 or 4 for a prettier final image. This will save you some time along the way.

5.14 `Scene::TraceRay`

In `Scene::SaveImage` you will find the following code as the innermost logic for determining the color of a pixel:

```
// Trace a ray from the camera toward the given direction
// to figure out what color to assign to this pixel.
pixel.color = TraceRay(
```

```
        camera,
        direction,
        ambientRefraction,
        fullIntensity,
        0);
```

`TraceRay` embodies the very core of this ray tracer's optical calculation engine. Four chapters will be devoted to how it simulates the physical behavior of light interacting with matter, covering the mathematical details of reflection and refraction. For now, all you need to know is that this function returns a color value that should be assigned to the given pixel based on light coming from that point on an object's surface.

5.15 Saving the final image to disk

After adding up the contributions of reflection from a point and refraction through that point, the function `Scene::TraceRay` returns the value of `colorSum` to the caller, `Scene::SaveImage`. This color value is used as the color of the pixel associated with the camera ray that intersected with the given point. `SaveImage` continues tracing rays from the camera to figure out the color of such pixels that have a surface intersection point along their line of sight. When `SaveImage` has explored every pixel in the image and has determined what colors they all should have, it performs three post-processing steps before saving an image to disk:

1. Sometimes tracing the ray through a pixel will lead to an ambiguity where multiple intersections are tied for being closest to a vantage point. The vantage point causing the problem may be the camera itself, or it might be a point on some object that the light ray reflects from or refracts through. To avoid complications that will be explained later, such pixels are marked and skipped in the ray tracing phase. In post-processing, `SaveImage` uses surrounding pixel colors to approximate what the ambiguous pixel's color should be.

2. If `antiAliasFactor` is greater than one, `SaveImage` averages square blocks of pixels into a single pixel value, as discussed in the earlier section on anti-aliasing. This reduces both the pixel width and the pixel height by the `antiAliasFactor`.

3. The floating point color values of each pixel, having an unpredictable range of values, must be converted to integers in the range 0 to 255 in order to be saved in PNG format. `SaveImage` searches the entire image for the maximum value of any red, green, or blue color component. That maximum floating point value (let's call it `M`) is used to linearly scale the color components of every pixel to fit in the range 0 to 255:

   ```
   R = round(255 * r / M)
   G = round(255 * g / M)
   B = round(255 * b / M)
   ```

   where `r`, `g`, `b` are floating point color components, and `R`, `G`, `B` are integer values in the range of a byte, or 0 to 255.

Unlike the pseudocode above, the C++ code performs the second calculation using a function called `ConvertPixelValue`, which has logic that ensures the result is always clamped to the inclusive range 0 to 255:

```
// Convert to integer red, green, blue, alpha values,
// all of which must be in the range 0..255.
rgbaBuffer[rgbaIndex++] = ConvertPixelValue(sum.red,   maxColorValue);
rgbaBuffer[rgbaIndex++] = ConvertPixelValue(sum.green, maxColorValue);
rgbaBuffer[rgbaIndex++] = ConvertPixelValue(sum.blue,  maxColorValue);
rgbaBuffer[rgbaIndex++] = OPAQUE_ALPHA_VALUE;
```

As a quirk in the `rgbaBuffer`, each pixel requires a fourth value called *alpha*, which determines how transparent or opaque the pixel is. PNG files may be placed on top of some background image such that they mask some parts of the background but allow other parts to show through. This ray tracing code makes all pixels completely opaque by setting all alpha values to `OPAQUE_ALPHA_VALUE` (or 255).

After filling in `rgbaBuffer` with single-byte values for red, green, blue, and alpha for each pixel in the image (a total of 4 bytes per pixel), `SaveImage` calls Lode Vandevenne's LodePNG code to write the buffer to a PNG file:

```
// Write the PNG file
const unsigned error = lodepng::encode(
    outPngFileName,
    rgbaBuffer,
    pixelsWide,
    pixelsHigh);
```

You now have a general idea of how a scene is broken up into pixels and converted into an image that can be saved as a PNG file. The code traces rays of light from the camera point toward thousands of vector directions, each time calling `TraceRay`. That function figures out what color to assign to the pixel associated with that direction from the camera point. In upcoming chapters we will explore `TraceRay` in detail to understand how it computes the intricate behavior of light as it interacts with physical matter. But first we will study an example of solving the mathematical problem of finding an intersection of a ray with a simple shape, the sphere.

Chapter 6

Sphere Intersections

6.1 General approach for all shapes

How do we find the intersections of a ray through space and a solid object? This is a separate algebra problem that we must solve for each different kind of shape. Each shape will then be coded in C++ as a separate class derived from `class SolidObject`. Each derived class will include the mathematical solution for the intersections of a ray with the given shape in its `AppendAllIntersections` method. In this chapter I cover an example of finding the solution for where a ray intersects with a sphere. In later chapters I will use a similar approach to show you how to write code for other shapes' `AppendAllIntersections` member functions. Here is an outline of the steps we will follow in every case.

1. Write a parametric equation for any point on the given ray. This means the ray equation will use a scalar parameter $u > 0$ to select any point along the ray.

2. Write one or more equations that describe the surfaces of the solid.

3. Algebraically solve the system of equations from steps 1 and 2. The solution will provide us with the set of all points that are both on the surfaces of the solid object and along the ray, i.e., the set of intersection points.

4. For each intersection, compute a unit vector that is normal (perpendicular) to the surface and pointing outward from the object.

5. Fill in an `Intersection` struct with the location of each intersection point, the square of the distance between the intersection point and the vantage point, and the surface normal unit vector at that point. Each `AppendAllIntersections` member function also must copy a pointer to the intersected object into the struct member `solid`, which is usually coded as

```
intersection.solid = this;
```

6. Append the `Intersection` struct to the output parameter `intersectionList`.

6.2 Parametric ray equation

The first step, writing an equation for any point along the ray, is identical for every shape, so it is critical to understand. Fortunately, it is very simple, both conceptually and algebraically. I say the equation is *parametric* because it uses a parameter variable u that indicates where along the ray a point is. By setting the value of u to any positive number, the equation calculates the x, y, and z components of a point on the ray. Let $\mathbf{D}$ be a vantage point, which is just a position vector for a fixed point in space. Let $\mathbf{E}$ be a direction vector pointing which way a ray emanates from the vantage point $\mathbf{D}$. It is important to understand that the ray extends infinitely in the direction of $\mathbf{E}$ from the vantage point $\mathbf{D}$, even though $\mathbf{E}$ is a vector with a finite magnitude.

I will write the parametric equation for a point on the ray three different ways, and explain each, just to make sure the concept is clear. The equation is most concisely expressed in vector form:

$$\mathbf{P} = \mathbf{D} + u\mathbf{E} \quad : \quad u > 0$$

where $\mathbf{P}$ is the position vector for any point along the ray. Note that $\mathbf{P}$, $\mathbf{D}$, and $\mathbf{E}$ are all vectors, but u is a scalar. Also note the constraint that u must be a positive real numer. If we let u be zero, $\mathbf{P}$ would be at the same location as $\mathbf{D}$, the vantage point. We want to exclude the vantage point from any intersections we find, primarily to prevent us from thinking a point on a surface casts a shadow on itself. We don't allow negative values for u because we would then be calculating point coordinates that are in the opposite direction as intended.

Another way to express the parametric line equation is by explicitly writing all the vector components in a single equation:

$$(P_x, P_y, P_z) = (D_x, D_y, D_z) + u(E_x, E_y, E_z)$$

Or equivalently, we can write a system of three linear scalar equations:

$$\begin{cases} P_x = D_x + uE_x \\ P_y = D_y + uE_y \\ P_z = D_z + uE_z \end{cases}$$

This last form makes it especially clear how, given fixed values for the vantage point (D_x, D_y, D_z) and the direction vector (E_x, E_y, E_z), we can adjust u to various values to select any point (P_x, P_y, P_z) along the ray. Simply put, the point $\mathbf{P} = (P_x, P_y, P_z)$ is a function of the parameter u. Conversely, if $\mathbf{P}$ is an arbitrary point along the ray, there is a unique value of u such that $\mathbf{P} = \mathbf{D} + u\mathbf{E}$. In order to find all intersections with a ray and a surface, our goal is to find all positive values of u such that $\mathbf{P}$ is on that surface.

6.3 Surface equation

Now we are ready to take the second step in the solution strategy: writing a list of equations for all the surfaces on the solid object. In this example, where the solid is a sphere, there is a single equation that must be satisfied by any point on the spherical surface:

$$(P_x - C_x)^2 + (P_y - C_y)^2 + (P_z - C_z)^2 = R^2$$

where $\mathbf{P} = (P_x, P_y, P_z)$ is any point on the sphere (intentionally the same $\mathbf{P}$ as in the parametric ray equation above), $\mathbf{C} = (C_x, C_y, C_z)$ is the center of the sphere expressed as a position vector, and R is the radius of the sphere.

6.4 Intersection of ray and surface

Step 3 in the solution strategy is to substitute the parametric ray equation into the surface equation(s). This means solving for all points $\mathbf{P}$ that are both on the ray and on one of the solid's surfaces. So we have

$$(P_x - C_x)^2 + (P_y - C_y)^2 + (P_z - C_z)^2 = R^2$$

with

$$\begin{cases} P_x = D_x + uE_x \\ P_y = D_y + uE_y \\ P_z = D_z + uE_z \end{cases}$$

Everywhere we see P_x, P_y, or P_z in the first equation, we substitute the appropriate right-hand side from the second system of linear equations. This is the key: we end up with a single equation with a single unknown, the parameter u.

$$\big((D_x + uE_x) - C_x\big)^2 + \big((D_y + uE_y) - C_y\big)^2 + \big((D_z + uE_z) - C_z\big)^2 = R^2$$

Expanding the squared expressions, we have

$$\begin{aligned}
&E_x^2u^2 + 2E_x(D_x - C_x)u + (D_x - C_x)^2 + \\
&E_y^2u^2 + 2E_y(D_y - C_y)u + (D_y - C_y)^2 + \\
&E_z^2u^2 + 2E_z(D_z - C_z)u + (D_z - C_z)^2 = R^2
\end{aligned}$$

We can write this as a standard-form quadratic equation in u:

$$\begin{aligned}
&(E_x^2 + E_y^2 + E_z^2)u^2 + \\
&2\big(E_x(D_x - C_x) + E_y(D_y - C_y) + E_z(D_z - C_z)\big)u + \\
&(D_x - C_x)^2 + (D_y - C_y)^2 + (D_z - C_z)^2 - R^2 \\
&= 0
\end{aligned}$$

Like any other quadratic equation in a single unknown variable, we have a constant coefficient for the u^2 term, another for the u term, and a constant term, all adding up to zero. If we create temporary symbols a, b, c for the coefficients, letting

$$\begin{cases}
a = E_x^2 + E_y^2 + E_z^2 \\
b = 2\big(E_x(D_x - C_x) + E_y(D_y - C_y) + E_z(D_z - C_z)\big) \\
c = (D_x - C_x)^2 + (D_y - C_y)^2 + (D_z - C_z)^2 - R^2
\end{cases}$$

We then have the familiar quadratic equation

$$au^2 + bu + c = 0$$

for which the solutions are

$$u = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Depending on the values of a, b, and c, the quadratic equation may have 0, 1, or 2 real solutions for u. These three special cases correspond to three different scenarios where a ray may miss the sphere entirely (0 solutions), may graze the sphere tangentially at one point (1 solution), or may pierce the sphere, entering at one point and exiting at another (2 solutions). The value of the expression inside the quadratic solution's square root (called the *radicand*) determines which of the three special cases we counter:

$$\begin{cases}
b^2 - 4ac < 0 & \text{0 solutions} \\
b^2 - 4ac = 0 & \text{1 solution} \\
b^2 - 4ac > 0 & \text{2 solutions}
\end{cases}$$

Interestingly, and usefully for our programming, we can express the values of a, b, c more concisely using vector dot products and magnitudes:

$$\begin{aligned}
a &= |\mathbf{E}|^2 \\
b &= 2\mathbf{E} \cdot (\mathbf{D} - \mathbf{C}) \\
c &= |\mathbf{D} - \mathbf{C}|^2 - R^2
\end{aligned}$$

In the C++ code for the function `Sphere::AppendAllIntersections`, $\mathbf{E}$ is the function parameter `direction`, and $\mathbf{D}$ is the parameter `vantage`. The center of the sphere is inherited from the base class `SolidObject`; we obtain its vector value via the function call `Center()`. The radius of the sphere is stored in `Sphere`'s member variable `radius`. So we express the calculation of the quadratic's coefficients and the resulting radicand value as:

```
const Vector displacement = vantage - Center();
const double a = direction.MagnitudeSquared();
const double b = 2.0 * DotProduct(direction, displacement);
const double c = displacement.MagnitudeSquared() - radius*radius;
const double radicand = b*b - 4.0*a*c;
```

Once we calculate `radicand`, we check to see if it is less than zero. If so, we immediately know that there are no intersections with this ray and the sphere. Otherwise, we know we can take the square root of the radicand, and proceed to calculate two values for u. (Because it is a very rare case, we don't worry about whether the radicand is zero. Whether it is zero or positive, we calculate separate values for u using the $\pm$ in the quadratic solution formula. It is faster, and harmless, to assume that there are two intersections, even if they are occasionally the same point in space.)

But it is not enough that a value of u is real; we must check each u value to see if it is positive. As stated before, a negative u value means the intersecion point lies in the wrong direction, as shown in Figure 6.1.

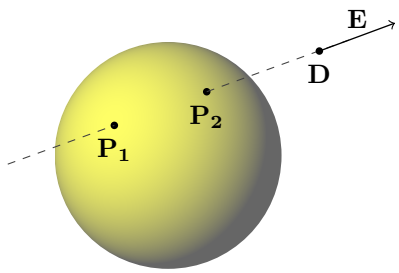

Figure 6.1: When $u < 0$, it means the intersection is in the opposite of the intended direction $\mathbf{E}$ from the vantage point $\mathbf{D}$. In this example, there are two intersections that lie in the wrong direction from $\mathbf{D}$: At $\mathbf{P_1}$, the value of u is -2.74, and at $\mathbf{P_2}$, $u = -1.52$.

In short, if u is zero, the intersection is at the vantage point, if u is negative, it is behind the vantage point, and if u is positive, it is in front of the vantage point. So only positive values of u count as valid intersections. When u is positive ($u > 0$), the larger the value of u, the farther away the intersection point is from the vantage point. Because of floating point rounding errors, we actually make sure that u is greater than a small constant value `EPSILON`, not zero. This prevents erroneous intersections from being found too close to the vantage point, which would cause problems when calculating shadows and lighting. `EPSILON` is defined as 10^{-6}, or one-millionth, in `imager.h`:

```
const double EPSILON = 1.0e-6;
```

If we find positive real solutions for u, we can plug each one back into the parametric ray equation to obtain the location of the intersection point:

$$\mathbf{P} = \mathbf{D} + u\mathbf{E}$$

In C++ code, this looks like:

```
intersection.point = vantage + u*direction;
```

The overloaded operators `*` and `+` for `class Vector` allow us to write this calculation in a natural and concise way, and because these operators are implemented as inline functions (in `imager.h`) they are very efficient in release/optimized builds.

6.5 Surface normal vector

Step 4 of the solution strategy for any solid object is to calculate the surface normal unit vector at each intersection point. This unit vector will be needed for calculating the effect of any light sources shining on that point at the object's surface. In the case of a sphere, the surface normal unit vector points exactly away from the center of the sphere, as shown in Figure 6.2.

If $\mathbf{Q}$ is the center of the sphere, $\mathbf{P}$ is the intersection point on the sphere's surface, and $\hat{\mathbf{n}}$ is the surface normal unit vector, then $\mathbf{P} - \mathbf{Q}$ is a vector that points in the same direction as $\hat{\mathbf{n}}$. We can divide the vector difference $\mathbf{P} - \mathbf{Q}$ by its magnitude $|\mathbf{P} - \mathbf{Q}|$ to convert it to the unit vector $\hat{\mathbf{n}}$:

$$\hat{\mathbf{n}} = \frac{\mathbf{P} - \mathbf{Q}}{|\mathbf{P} - \mathbf{Q}|}$$

In C++ code, we can use the method `UnitVector` in `class Vector` to do the same thing:

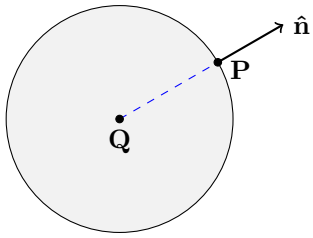

Figure 6.2: The surface normal vector $\hat{\mathbf{n}}$ at any point $\mathbf{P}$ on the surface of a sphere points away from the sphere's center $\mathbf{Q}$.

```
intersection.surfaceNormal =
    (intersection.point - Center()).UnitVector();
```

6.6 Filling in the `Intersection` struct

Step 5 has been mostly accomplished already — we have calculated the intersection point and the surface normal unit vector, and both are stored in the `struct Intersection` variable called `intersection`. But we also need to store the square of the distance between the intersection point and the vantage point inside `intersection`. The distance squared calculation and the intersection point calculation both involve the term $u\mathbf{E}$ (or `u * direction` in C++), so it is a bit more efficient to calculate the product once and store it in a temporary variable of type `Vector`. After filling out all the fields of the local variable `intersection`, we must remember to insert it into the output list parameter `intersectionList`:

```
intersectionList.push_back(intersection);
```

6.7 C++ sphere implementation

The entire function `Sphere::AppendAllIntersections` is shown here, just as it appears in `sphere.cpp`:

```
void Sphere::AppendAllIntersections(
    const Vector& vantage,
    const Vector& direction,
    IntersectionList& intersectionList) const
{
    // Calculate the coefficients of the quadratic equation
    //     au^2 + bu + c = 0.
    // Solving this equation gives us the value of u
    // for any intersection points.
    const Vector displacement = vantage - Center();
    const double a = direction.MagnitudeSquared();
    const double b = 2.0 * DotProduct(direction, displacement);
    const double c = displacement.MagnitudeSquared() - radius*radius;

    // Calculate the radicand of the quadratic equation solution formula.
    // The radicand must be non-negative for there to be real solutions.
    const double radicand = b*b - 4.0*a*c;
    if (radicand >= 0.0)
    {
        // There are two intersection solutions, one involving
        // +sqrt(radicand), the other -sqrt(radicand).
        // Check both because there are weird special cases,
        // like the camera being inside the sphere,
        // or the sphere being behind the camera (invisible).
        const double root = sqrt(radicand);
        const double denom = 2.0 * a;
        const double u[2] = {
            (-b + root) / denom,
            (-b - root) / denom
        };

        for (int i=0; i < 2; ++i)
```

```
        {
            if (u[i] > EPSILON)
            {
                Intersection intersection;
                const Vector vantageToSurface = u[i] * direction;
                intersection.point = vantage + vantageToSurface;

                // The normal vector to the surface of
                // a sphere is outward from the center.
                intersection.surfaceNormal =
                    (intersection.point - Center()).UnitVector();

                intersection.distanceSquared =
                    vantageToSurface.MagnitudeSquared();

                intersection.solid = this;
                intersectionList.push_back(intersection);
            }
        }
    }
}
```

6.8 Switching gears for a while

We will look at the algebraic solutions for other shapes in later chapters, but first we need to take a look at how these lists of intersection objects are used to help implement an optical simulator. We have already discussed how the ray tracing algorithm looks for intersections between camera rays and solid objects. In the next chapter we begin exploring how the ray tracing algorithm assigns a color to an intersection it finds, based on optical processes like reflection and refraction.

Chapter 7

Optical Computation

7.1 Overview

We have already studied how `Scene::SaveImage` calls `Scene::TraceRay` to trace rays from the camera point out into the scene. `TraceRay` determines whether the ray intersects with the surface of any of the objects in the scene. If `TraceRay` finds one or more intersections along a given direction, it then picks the closest intersection and calls `Scene::CalculateLighting` to determine what color to assign to the pixel corresponding to that intersection. This is the first of four chapters that discuss the optical computations implemented by `CalculateLighting` and other functions called by it.

`CalculateLighting` is the heart of the ray tracing algorithm. Given a point on the surface of some solid, and the direction from which it is seen, `CalculateLighting` uses physics equations to decide where that ray bounces, bends, and splits into multiple rays. It takes care of all this work so that the different classes derived from `SolidObject` can focus on surface equations and point containment. Because instances of `SolidObject` provide a common interface for finding intersections with surfaces of any shape, `CalculateLighting` can treat all solid objects as interchangeable parts, applying the same optical concepts in every case.

`CalculateLighting` is passed an `Intersection` object, representing the intersection of a camera ray with the point on the surface that is closest to the camera in a particular direction from the vantage point. Its job is to add up the contributions of all light rays sent toward the camera from that point and to return the result as a `Color` struct that contains separate red, green, and blue values. It calls the following helper functions to determine the contributions to the pixel color caused by different physical processes:

- `CalculateMatte` : Calculates the *matte reflection* at the intersection point, which is the part of the color caused by soft scattering of light by materials like paper or chalk. Matte reflection diffuses incident light in all directions equally.

- `CalculateReflection` : Calculates the *mirror reflection* at the point, meaning the shiny reflection of surrounding objects from a material like glass or silver. Mirror reflection causes a reflected light ray to bounce from the object at an equal but opposite angle as the incident ray, not in all directions like matte reflection does. Mirror reflection is composed of both *glossy reflection* due to an imaginary coating on the object which can have any desired color, and *refractive reflection* that reflects light of all colors equally. Both types of reflection are added together to produce a composite mirror-reflected image. This is explained in more detail below.

- `CalculateRefraction` : Calculates the *refraction* (or lensing) of light through transparent materials like water or glass. Although the light passes right through a transparent substance, the rays of light are bent from their original straight path. Transparent objects with curved surfaces cause other objects seen through them to appear distorted in size and shape.

7.2 `class Optics`

Before diving into the details of the reflection and refraction code, it is important to understand how this C++ code represents the optical properties of a point on the surface of a solid object. Take a look in `imager.h` for the class called `Optics`. This class holds a matte color, a gloss color, and an opacity value.

7.2.1 Matte color

The matte color indicates the intrinsic color scattered equally in all directions by that point when perfectly white light shines on it. In the real world, actual white light is composed of an infinite number of wavelengths (all the colors of the spectrum), but in our simple model we use red, green, and blue, the primary colors as perceived by the human eye. The matte color represents what fractions of red, green, and blue light the point reflects, irrespective of what color light reaches that point on the object. If you think of a red ball being inherently red whether or not any light is shining on it (philosophical musings notwithstanding), then you get the idea here. Pure red light shining on a pure red object will look the same as white light shining on a red object or red light shining on a white object: in all three cases, only red light reaches the viewer's eye. However, a white object and a red object will appear very different from each other when green light is shining on both of them. The matte color is a property of a point on an object independent of any illumination reaching that point. (We will soon see how the inherent colors of an object are combined with illumination to produce a perceived color.)

7.2.2 Gloss color

Similarly, the gloss color indicates the fraction of red, green, and blue light that bounces from the point due to glossy reflection. The gloss color provides a way to make mirror images of surrounding scenery have a different color tone than the original scenery. The gloss color is independent of the matte color, so as an example, an object may have a reddish matte color while simultaneously reflecting blue mirror images of nearby objects. If the gloss color is white (that is, the red, green, and blue color components are all 1), there is no color distortion of reflected scenery — the mirror image of an object will have the same color as the original.

7.2.3 Avoiding amplification

The constructor for the `Optics` class and its member functions `SetGlossColor` and `SetMatteColor` enforce that every matte color and gloss color have red, green, and blue color components constrained to the range 0 to 1. Any attempt to go outside this range results in an exception. This constraint is necessary (but not sufficient) to prevent unrealistic amplification of light; we don't want an object to reflect more light than it receives from the environment, because that can cause images to look wrong. Amplification can also lead to excessive run time or crashes due to floating point overflows after a long series of amplifying reflections. This undesirable amplification problem can also occur if matte and gloss color components add up to values greater than 1. No such constraint is enforced by the `Optics` class, but that class does provide the member function `SetMatteGlossBalance` to help you avoid this problem. Call this function with the first parameter `glossFactor` in the range 0 to 1 to select how much of the opaque behavior should be due to matte reflection and how much should be due to glossy reflection. A gloss factor of 1 causes glossy reflection to dominate completely with no matte reflection at all, while a gloss factor of 0 eliminates all glossy reflection leaving only matte reflection present. `SetMatteGlossBalance` will store an adjusted gloss color and matte color inside the `Optics` instance that are guaranteed not to cause light amplification, while preserving the intended tone of each color.

7.2.4 Opacity

The opacity value is a number in the range 0 to 1 that tells how opaque or transparent that point on the object's surface is. If the opacity is 1, it means that all light shining on that point is prevented from entering the object: some is matte-reflected, some is gloss-reflected, and the rest is absorbed by the point. If the opacity is 0, it means that point on the object's surface is

completely transparent in the sense that it has no matte or glossy reflection. In this case, the matte color and gloss colors are completely ignored; their values have no effect on the color of the pixel when the opacity is 0.

However, there is a potentially confusing aspect of the opacity value. As mentioned above, there are two types of mirror-like reflection: glossy reflection and refractive reflection. Even when the opacity is 0, the surface point can still exhibit mirror reflection as a side-effect of refraction. This is because in the real world, refraction and reflection are physically intertwined phenomena; they are both continuously variable depending on the angle between incident light and a line perpendicular to the surface. However, glossy reflection is necessary in addition to refractive reflection in any complete ray tracing model for two reasons: refractive reflection does not allow any color tinge to be added to reflective images, and more importantly, refractive reflection does not provide for surfaces like silver or polished steel that reflect almost all incident light (at least not without making light travel faster than it does in a vacuum, which is physically impossible). In short, the real world has more than one mechanism for creating mirror images, and this ray tracing code therefore emulates them for the sake of generating realistic images.

7.2.5 Splitting the light's energy

For now, the following very rough pseudo-code formulas summarize how the energy in a ray of light is distributed across matte reflection, mirror reflection, and refraction:

```
matte   = opacity*matteColor
mirror  = opacity*glossColor + (1-opacity)*(1-transmitted)
refract = (1-opacity)*transmitted
```

In these formulas, `opacity`, `matteColor`, and `glossColor` are the values stored in an instance of the `Optics` class. As `opacity` increases from 0 to 1, the value of `(1-opacity)` decreases from 1 to 0. You can think of `(1-opacity)` as being the degree of transparency of the surface point. The symbol `transmitted` represents the fraction of light that penetrates the surface of the object and continues on through its interior. The value of `(1-transmitted)` is the remainder of the light intensity that contributes the refractive part of mirror reflection. Again I stress that these formulas are rough. The actual C++ code is more complicated, but we will explore that in detail later. For now, the intent is to convey how a ray of light is split up into an opaque part and a transparent part, with the somewhat counterintuitive twist that some of the "transparent" part consists of refractively-reflected light that doesn't actually pass through the object. You would set `opacity` equal to 0 to model a substance like clear glass or water, 1 to model an opaque substance like wood or steel, and somewhere in between 0 and 1 for dusty glass.

7.2.6 Surface optics for a solid

The member function `SolidObject::SurfaceOptics` returns a value of type `Optics` when given the location of any point on the object's surface. By default, solid objects have uniform optics across their entire surface, but derived classes may override the `SurfaceOptics` member function to make opacity, matte color, and gloss color vary depending on where light strikes the object's surface. (See `class ChessBoard` in `chessboard.h` and `chessboard.cpp` for an example of variable optics.)

7.3 Implementation of `CalculateLighting`

7.3.1 Recursion limits

The `CalculateLighting` member function participates in mutual recursion with the other member functions `CalculateRefraction` and `CalculateReflection`, although indirectly. `TraceRay` calls `CalculateLighting`, which in turn calls the other member functions `CalculateRefraction` and `CalculateReflection`, both of which make even deeper calls to `TraceRay` as needed. There are two built-in limits to this mutual recursion, for the sake of efficiency and to avoid stack overflow crashes from aberrent special cases. The first is a limit on how many times `CalculateLighting` may be recursively called:

```
if (recursionDepth <= MAX_OPTICAL_RECURSION_DEPTH)
```

The recursion depth starts out at 0 when `SaveImage` calls `TraceRay`. `TraceRay` adds 1 to the recursion depth that it passes to `CalculateLighting`. The other functions that participate in mutual recursion, `CalculateLighting`, `CalculateRefraction`, and `CalculateReflection`, all pass this recursion depth along verbatim. As the recursion gets deeper and deeper, each nested call to `TraceRay` keeps increasing the recursion depth passed to the other functions. Once the recursion depth exceeds the maximum allowed recursion depth, `CalculateLighting` bails out immediately and returns a pure black color to its caller. This is not the most elegant behavior, but it should happen only in extremely rare circumstances. It is certainly better than letting the program crash from a stack overflow and not getting any image at all!

The second recursion limiter is the ray intensity. As a ray is reflected or refracted from object to object in the scene, it becomes weaker as these optical processes reduce its remaining energy. It is possible for the ray of light to become so weak that it has no significant effect on the pixel color. The ray intensity is calculated and passed down the chain of recursive calls. The `CalculateLighting` function avoids excessive calculation for cases where the light is too weak to matter via the following conditional statement:

```
if (IsSignificant(rayIntensity))
```

This helper function is very simple. It returns `true` only if one of the components of the color parameter is large enough to matter for rendering. The threshold for significance is a factor of one in a thousand, safely below the $\frac{1}{256}$ color resolution of PNG and other similar image formats:

```
const double MIN_OPTICAL_INTENSITY = 0.001;

inline bool IsSignificant(const Color& color)
{
    return
        (color.red   >= MIN_OPTICAL_INTENSITY) ||
        (color.green >= MIN_OPTICAL_INTENSITY) ||
        (color.blue  >= MIN_OPTICAL_INTENSITY);
}
```

7.3.2 Ambiguous intersections

The `TraceRay` member function calls `FindClosestIntersection` to search for an intersection in the given direction from the vantage point. That helper function is located in `scene.cpp` and is coded like this:

```
int Scene::FindClosestIntersection(
    const Vector& vantage,
    const Vector& direction,
    Intersection& intersection) const
{
    // Build a list of all intersections from all objects.
    cachedIntersectionList.clear();     // empty any previous contents
    SolidObjectList::const_iterator iter = solidObjectList.begin();
    SolidObjectList::const_iterator end  = solidObjectList.end();
    for (; iter != end; ++iter)
    {
        const SolidObject& solid = *(*iter);
        solid.AppendAllIntersections(
            vantage,
            direction,
            cachedIntersectionList);
    }
    return PickClosestIntersection(cachedIntersectionList, intersection);
}
```

`FindClosestIntersection` iterates through the solid objects in the scene and collects a list of all intersections the ray may have with them. Then it passes this list to another function called `PickClosestIntersection` to find the intersection closest to the vantage point, if any intersection exists. That function returns 0 if there is no intersection, or 1 if it finds a single intersection that is closer than all others. In the first case, `TraceRay` knows that the ray continues on forever without hitting anything, and thus it can use the scene's background color for the ray. In the second case, the closest intersection it found is passed to `CalculateLighting` to evaluate how the ray of light behaves when it strikes that point. However, there is another case:

there can be a tie for the closest intersection point, meaning that the closest intersection point overlaps with more than one surface. The surfaces may belong to different solid objects, or they may belong to a single object. For example, it is possible for a ray of light to strike the exact corner point on a cube. This would register as three separate intersections, one for each of the three cube faces that meet at that corner point. In a case like this, `PickClosestIntersection` returns 3, meaning there is a three-way tie for the closest intersection. As you can see in the code listing above, `FindClosestIntersection` returns whatever integer value it receives back from its call to `PickClosestIntersection`.

As mentioned earlier in this chapter, `CalculateLighting` determines three different kinds of optical behavior: matte reflection, mirror reflection, and refraction. All three of these physical phenomena depend on the angle of the indicent ray of light with the surface that the ray strikes. If there is more than one surface associated with the closest intersection point, we run into complications. The behavior of the ray of light becomes ambiguous when we don't know which surface to use. One might think we could just split the ray of light into equal parts and let the resulting rays reflect or refract off all of the surfaces independently. One problem with this approach is that it includes surfaces that in reality would not experience the ray of light in the first place. Consider the case of the cube corner mentioned above. It is possible for the incident ray to hit the corner in such a way that only two of the faces are actually lit, the third being in shadow.

Even more complicated is the case of two solids touching at a point where the ray of light strikes. For example, imagine two cubes with corners exactly touching. It becomes quite unclear what should happen when light strikes this point. To avoid these complications, `TraceRay` doesn't even try to figure out what to do with the light ray. Instead, it throws an exception that indicates that it has encountered an ambiguous intersection. When this exception occurs, it is caught by `SaveImage`, which marks the associated pixel as being in an ambiguous state. `SaveImage` then carries on tracing rays for other pixels in the image. After it has completed the entire image, it goes back and patches up any ambiguous pixels by averaging the colors of surrounding pixels that weren't marked ambiguous. Although this is not a perfect solution, it does a surprisingly good job at hiding the pixels that were skipped over, and avoids a lot of unpleasant special cases.

7.3.3 Debugging support

Sometimes when you are creating a new image, you will run across situations you don't understand. The resulting image will not look as you expect. You might look at the output and ask something like, "why is this pixel so much brighter than that pixel?" Because there are so many thousands of pixels, tracing through with the debugger can be tedious. Many debuggers support conditional breakpoints, allowing you to run the code until a particular pixel is reached. However, doing this can really slow down the run time to the point that it becomes unusuable in practice. (Microsoft Visual Studio appears to be re-parsing the conditional breakpoint every time, making it take longer than I have ever been willing to wait.)

To assist in understanding how the code is treating one or more particular pixels in an image, it is possible to change a preprocessor directive inside `imager.h` so as to enable some extra debugging code in `scene.cpp`. Just find the line toward the top of that header file that looks like this:

```
#define RAYTRACE_DEBUG_POINTS 0
```

Change that line to the following and rebuild the entire project to enable the debugger feature:

```
#define RAYTRACE_DEBUG_POINTS 1
```

Then in the code that builds and renders a scene, call the member function `AddDebugPoint` for each pixel you are interested in. Just pass the horizontal and vertical coordinates of each pixel. Here is an example of what these calls might look like:

```
scene.AddDebugPoint(419, 300);
scene.AddDebugPoint(420, 300);
scene.AddDebugPoint(421, 300);
```

How do you know the exact coordinates of the pixels? In Windows, I use the built-in Paint utility (`mspaint.exe`) to edit the output PNG file. Then I move the mouse cursor until it is over a pixel I am interested in. At the bottom of the window it shows the coordinates of the pixel. One important tip to keep in mind: when you are debugging, it is easiest to disable anti-aliasing by making the anti-alias factor be 1. Otherwise you will have to multiply all of your horizontal and vertical pixel coordinates by the anti-aliasing factor to figure out what values should be passed to `AddDebugPoint`. Even then, the behavior you are trying to figure out for a particular pixel in the output image can be spread across a grid of pre-anti-aliased pixels, causing more complication. So for that reason also, it is far less confusing to debug with anti-aliasing turned off.

If you search for `RAYTRACE_DEBUG_POINTS` in `scene.cpp`, you will see a few places where it is used to conditionally include or exclude blocks of source code. Most of these code blocks check to see if the member variable `activeDebugPoint` is set to a non-null value, and if so, they print some debug information. That member variable is set by the following conditional code block in `Scene::SaveImage`, which searches through `debugPointList` for pixel coordinates that match those of the current pixel. Whenever you call `AddDebugPoint` on a scene object, it adds a new entry to `debugPointList`.

```
#if RAYTRACE_DEBUG_POINTS
        {
            using namespace std;

            // Assume no active debug point unless we find one below.
            activeDebugPoint = NULL;

            DebugPointList::const_iterator iter = debugPointList.begin();
            DebugPointList::const_iterator end  = debugPointList.end();
            for(; iter != end; ++iter)
            {
                if ((iter->iPixel == i) && (iter->jPixel == j))
                {
                    cout << endl;
                    cout << "Hit breakpoint at (";
                    cout << i << ", " << j <<")" << endl;
                    activeDebugPoint = &(*iter);
                    break;
                }
            }
        }
#endif
```

The innermost `if` statement is an ideal place to set a debug breakpoint. The code will run at almost full speed, far faster than trying to use conditional breakpoints in your debugger. Even without the debugger, you may find that reading the debug output is sufficient to figure out why the ray tracer code is doing something unexpected.

Be sure to change the `#define` back to 0 when you are done debugging, to make the code run at full speed again, and to avoid printing out the extra debug text. Note that you do **not** need to remove your calls to `AddDebugPoint`; they will not harm anything nor cause the rendering to run any slower.

7.3.4 Source code listing

Below is a listing of the complete source code for `TraceRay` and `CalculateLighting`. The next three chapters explain the mathematics and algorithms of the important optical helper functions they call: `CalculateMatte`, `CalculateRefraction`, and `CalculateReflection`.

```
Color Scene::TraceRay(
    const Vector& vantage,
    const Vector& direction,
    double refractiveIndex,
    Color rayIntensity,
    int recursionDepth) const
{
    Intersection intersection;
    const int numClosest = FindClosestIntersection(
        vantage,
```

```
        direction,
        intersection);

    switch (numClosest)
    {
    case 0:
        // The ray of light did not hit anything.
        // Therefore we see the background color attenuated
        // by the incoming ray intensity.
        return rayIntensity * backgroundColor;

    case 1:
        // The ray of light struck exactly one closest surface.
        // Determine the lighting using that single intersection.
        return CalculateLighting(
            intersection,
            direction,
            refractiveIndex,
            rayIntensity,
            1 + recursionDepth);

    default:
        // There is an ambiguity: more than one intersection
        // has the same minimum distance.  Caller must catch
        // this exception and have a backup plan for handling
        // this ray of light.
        throw AmbiguousIntersectionException();
    }
}

// Determines the color of an intersection,
// based on illumination it receives via scattering,
// glossy reflection, and refraction (lensing).
// Determines the color of an intersection,
// based on illumination it receives via scattering,
// glossy reflection, and refraction (lensing).
Color Scene::CalculateLighting(
    const Intersection& intersection,
    const Vector& direction,
    double refractiveIndex,
    Color rayIntensity,
    int recursionDepth) const
{
    Color colorSum(0.0, 0.0, 0.0);

#if RAYTRACE_DEBUG_POINTS
    if (activeDebugPoint)
    {
        using namespace std;

        Indent(cout, recursionDepth);
        cout << "CalculateLighting[" << recursionDepth << "] {" << endl;

        Indent(cout, 1+recursionDepth);
        cout << intersection << endl;

        Indent(cout, 1+recursionDepth);
        cout << "direction=" << direction << endl;

        Indent(cout, 1+recursionDepth);
        cout.precision(4);
        cout << "refract=" << fixed << refractiveIndex;
        cout << ", intensity=" << rayIntensity << endl;

        Indent(cout, recursionDepth);
        cout << "}" << endl;
    }
#endif

    // Check for recursion stopping conditions.
    // The first is an absolute upper limit on recursion,
    // so as to avoid stack overflow crashes and to
    // limit computation time due to recursive branching.
```

```
    if (recursionDepth <= MAX_OPTICAL_RECURSION_DEPTH)
    {
        // The second limit is checking for the ray path
        // having been partially reflected/refracted until
        // it is too weak to matter significantly for
        // determining the associated pixel's color.
        if (IsSignificant(rayIntensity))
        {
            if (intersection.solid == NULL)
            {
                // If we get here, it means some derived class forgot to
                // initialize intersection.solid before appending to
                // the intersection list.
                throw ImagerException("Undefined solid at intersection.");
            }
            const SolidObject& solid = *intersection.solid;

            // Determine the optical properties at the specified
            // point on whatever solid object the ray intersected with.
            const Optics optics = solid.SurfaceOptics(
                intersection.point,
                intersection.context
            );

            // Opacity of a surface point is the fraction 0..1
            // of the light ray available for matte and gloss.
            // The remainder, transparency = 1-opacity, is
            // available for refraction and refractive reflection.
            const double opacity = optics.GetOpacity();
            const double transparency = 1.0 - opacity;
            if (opacity > 0.0)
            {
                // This object is at least a little bit opaque,
                // so calculate the part of the color caused by
                // matte (scattered) reflection.
                const Color matteColor =
                    opacity *
                    optics.GetMatteColor() *
                    rayIntensity *
                    CalculateMatte(intersection);

                colorSum += matteColor;

#if RAYTRACE_DEBUG_POINTS
                if (activeDebugPoint)
                {
                    using namespace std;

                    Indent(cout, recursionDepth);
                    cout << "matteColor=" << matteColor;
                    cout << ", colorSum=" << colorSum;
                    cout << endl;
                }
#endif
            }

            double refractiveReflectionFactor = 0.0;
            if (transparency > 0.0)
            {
                // This object is at least a little bit transparent,
                // so calculate refraction of the ray passing through
                // the point. The refraction calculation also tells us
                // how much reflection was caused by the interface
                // between the current ray medium and the medium it
                // is now passing into.  This reflection factor will
                // be combined with glossy reflection to determine
                // total reflection below.
                // Note that only the 'transparent' part of the light
                // is available for refraction and refractive reflection.

                colorSum += CalculateRefraction(
                    intersection,
                    direction,
```

```
                    refractiveIndex,
                    transparency * rayIntensity,
                    recursionDepth,
                    refractiveReflectionFactor  // output parameter
                );
            }

            // There are two sources of shiny reflection
            // that need to be considered together:
            // 1. Reflection caused by refraction.
            // 2. The glossy part.

            // The refractive part causes reflection of all
            // colors equally.  Each color component is
            // diminished based on transparency (the part
            // of the ray left available to refraction in
            // the first place).
            Color reflectionColor (1.0, 1.0, 1.0);
            reflectionColor *= transparency * refractiveReflectionFactor;

            // Add in the glossy part of the reflection, which
            // can be different for red, green, and blue.
            // It is diminished to the part of the ray that
            // was not available for refraction.
            reflectionColor += opacity * optics.GetGlossColor();

            // Multiply by the accumulated intensity of the
            // ray as it has traveled around the scene.
            reflectionColor *= rayIntensity;

            if (IsSignificant(reflectionColor))
            {
                const Color matteColor = CalculateReflection(
                    intersection,
                    direction,
                    refractiveIndex,
                    reflectionColor,
                    recursionDepth);

                colorSum += matteColor;
            }
        }
    }

#if RAYTRACE_DEBUG_POINTS
    if (activeDebugPoint)
    {
        using namespace std;

        Indent(cout, recursionDepth);
        cout << "CalculateLighting[" << recursionDepth << "] returning ";
        cout << colorSum << endl;
    }
#endif

    return colorSum;
}
```

Chapter 8

Matte Reflection

8.1 Source code listing

When `CalculateLighting` notices that a ray has intersected with a surface point that has some degree of opacity, it calls `CalculateMatte` to figure out the intensity and color of light scattered from that point. Let's take a look at the source code for `CalculateMatte`, followed by a detailed discussion of how it works.

```
// Determines the contribution of the illumination of a point
// based on matte (scatter) reflection based on light incident
// to a point on the surface of a solid object.
Color Scene::CalculateMatte(const Intersection& intersection) const
{
    // Start at the location where the camera ray hit
    // a surface and trace toward all light sources.
    // Add up all the color components to create a
    // composite color value.
    Color colorSum(0.0, 0.0, 0.0);

    // Iterate through all of the light sources.
    LightSourceList::const_iterator iter = lightSourceList.begin();
    LightSourceList::const_iterator end  = lightSourceList.end();
    for (; iter != end; ++iter)
    {
        // Each time through the loop, 'source'
        // will refer to one of the light sources.
        const LightSource& source = *iter;

        // See if we can draw a line from the intersection
        // point toward the light source without hitting any surfaces.
        if (HasClearLineOfSight(intersection.point, source.location))
        {
            // Since there is nothing between this point on the object's
            // surface and the given light source, add this light source's
            // contribution based on the light's color, luminosity,
            // squared distance, and angle with the surface normal.

            // Calculate a direction vector from the intersection point
            // toward the light source point.
            const Vector direction = source.location - intersection.point;

            const double incidence = DotProduct(
                intersection.surfaceNormal,
                direction.UnitVector()
            );

            // If the dot product of the surface normal vector and
            // the ray toward the light source is negative, it means
            // light is hitting the surface from the inside of the object,
            // even though we thought we had a clear line of sight.
            // If the dot product is zero, it means the ray grazes
            // the very edge of the object.  Only when the dot product
            // is positive does this light source make the point brighter.
            if (incidence > 0.0)
```

```
            {
                const double intensity =
                    incidence / direction.MagnitudeSquared();

                colorSum += intensity * source.color;
            }
        }
    }

    return colorSum;
}
```

8.2 Clear lines of sight

Because actual matte surfaces scatter incident light in all directions regardless of the source, an ideal ray tracing algorithm would search in the infinite number of directions from the point for any light that might end up there from the surroundings. One of the limitations of the C++ code that accompanies this book is that it does not attempt such a thorough search. There are two reasons for this: avoiding code complexity and greatly reducing the execution time for rendering an image. In our simplified model `CalculateMatte` looks for direct routes to point light sources only. It iterates through all point light sources in the scene, looking for unblocked lines of sight. To make this line-of-sight determination, `CalculateMatte` calls `HasClearLineOfSight`, a helper function that uses `FindClosestIntersection` to figure out whether there is any blocking point between the two points passed as arguments to it:

```
bool Scene::HasClearLineOfSight(
    const Vector& point1,
    const Vector& point2) const
{
    // Subtract point2 from point1 to obtain the direction
    // from point1 to point2, along with the square of
    // the distance between the two points.
    const Vector dir = point2 - point1;
    const double gapDistanceSquared = dir.MagnitudeSquared();

    // Iterate through all the solid objects in this scene.
    SolidObjectList::const_iterator iter = solidObjectList.begin();
    SolidObjectList::const_iterator end  = solidObjectList.end();
    for (; iter != end; ++iter)
    {
        // If any object blocks the line of sight,
        // we can return false immediately.
        const SolidObject& solid = *(*iter);

        // Find the closest intersection from point1
        // in the direction toward point2.
        Intersection closest;
        if (0 != solid.FindClosestIntersection(point1, dir, closest))
        {
            // We found the closest intersection, but it is only
            // a blocker if it is closer to point1 than point2 is.
            // If the closest intersection is farther away than
            // point2, there is nothing on this object blocking
            // the line of sight.

            if (closest.distanceSquared < gapDistanceSquared)
            {
                // We found a surface that is definitely blocking
                // the line of sight.  No need to keep looking!
                return false;
            }
        }
    }

    // We would not find any solid object that blocks the line of sight.
    return true;
}
```

Any light source with a clear line of sight adds to the intensity and color of the point based on its distance and angle of incidence, as described in detail below. We therefore miss light reflected

from other objects or lensed through other objects, but we do see realistic gradations of light and shadow across curved matte surfaces.

8.3 Brightness of incident light

When `CalculateMatte` finds a light source that has a clear line of sight to an intersection point, it uses two vectors to determine how slanted the light from that source is at the surface, which in turn determines how bright the light appears there. The first vector is the surface normal vector $\hat{\mathbf{n}}$, which is the unit vector pointing outward from the solid object at the intersection point, perpendicular to the surface. The second is $\hat{\lambda}$, a unit vector from the intersection point toward the light source. See Figure 8.1.

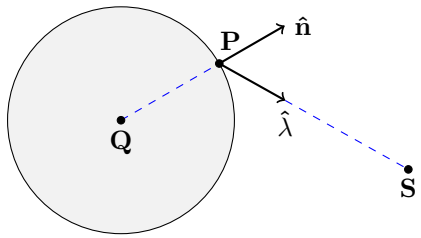

Figure 8.1: Matte reflection: a light source **S** illuminates the point **P** on the sphere with an intensity proportional to the dot product of the surface normal unit vector $\hat{\mathbf{n}}$ and the illumination unit vector $\hat{\lambda}$.

Then `CalculateMatte` calculates the dot product of these two vectors, $\hat{\mathbf{n}} \cdot \hat{\lambda}$, and assigns the result to the variable `incidence`:

```
const double incidence = DotProduct(
    intersection.surfaceNormal,
    direction.UnitVector()
);
```

Notice that this code explicitly converts `direction` to a unit vector, but assumes that the surface normal vector is already a unit vector; it is the responsibility of every class derived from `SolidObject` to fill in every intersection struct with a surface normal vector whose magnitude is 1.

As discussed in the earlier chapter about vectors, we can use the dot product of two unit vectors to determine how aligned the vectors are. There are three broad categories for the dot product value:

- **Dot product is negative**: The light source is shining on the inside surface of the solid, away from the camera's view. The portion of a light ray subject to matte reflection experiences the object as opaque, so none of the matte-scattered light travels through the object's surface. Any light shining on the inside surface is therefore assumed to be invisible to the camera, both because it should not have arrived there in the first place and because the object itself blocks the camera's view of interior surfaces.

- **Dot product is zero**: The light is grazing the surface exactly parallel to it. This causes the light to be spread out infinitely thin. This is the critical angle beyond which the surface point is in shade, and less than which the light makes a contribution to the illumination of that point.

- **Dot product is positive**: The light source makes the point on the object's surface brighter. The contribution of the light source is maximized when the dot product is equal to 1, which is the greatest possible value of the dot product of two unit vectors. This happens when the surface normal vector and the illumination vector are pointing in exactly the same direction, i.e., when the light is shining down on the surface point exactly perpendicular to the plane tangent to the surface at that point.

So this is why `CalculateMatte` has the conditional statement

```
if (incidence > 0.0)
```

When `incidence` is positive, we add a contribution from the given light source based on the inverse-square law and the incidence value:

```
const double intensity =
    incidence / direction.MagnitudeSquared();
```

So `intensity` holds a positive number that becomes larger as the light angle is closer to being perpendicular to the surface or as the light source gets nearer to the surface. Finally, the red, green, and blue color components of the local variable `colorSum` are updated based the intensity and color of the light source shining on the intersection point:

```
colorSum += intensity * source.color;
```

The C++ code uses the overloaded operator `*` to multiply the intensity (a double-precision floating point number) with the source color (a `Color` struct), to produce a result color whose red, green, and blue values have been scaled in proportion to the calculated intensity. This multiplication operator appears in `imager.h` as the line

```
inline Color operator * (double scalar, const Color &color)
```

The scaled color value is then added to `colorSum` using the overloaded operator `+=`, which is implemented as the following member function inside `struct Color`:

```
Color& operator += (const Color& other)
```

8.4 Using `CalculateMatte`'s return value

After iterating through all light sources, `CalculateMatte` returns `colorSum`, which is the weighted sum of all incident light at the given point. The caller, `CalculateLighting`, further adjusts this return value to account for the opacity factor, the inherent matte color of the surface point, and by the parameter `rayIntensity`.

This last multiplier, `rayIntensity`, was mentioned in the previous chapter. As we discussed there, and as we will explore in more detail in the next two chapters, mirror reflection and refraction can cause a ray of light that has been traced from the camera to split into multiple weaker rays that bounce in many directions throughout the scene. Each weaker ray can split into more rays, causing a branching tree of rays spreading in a large number of diverging directions. To handle this complexity, `CalculateLighting` calls `CalculateReflection` and `CalculateRefraction`, and those two functions can call back into `TraceRay` as needed, which in turn calls `CalculateLighting`. This means that `CalculateLighting` is indirectly recursive: it doesn't call itself directly, but it calls other functions that call back into it. Each time a ray is reflected or refracted, it generally becomes weaker, and so even though the total number of rays keeps increasing, they each make a decreasing contribution to the pixel's color components as seen by the camera.

The `rayIntensity` parameter provides the means for the rays getting weaker each time they are reflected or refracted. The indirectly-recursive calls take the existing `rayIntensity` value that was passed to them, and diminish it even further before calling back into `CalculateLighting`. It is important to understand that when `CalculateMatte` is called, its return value also needs to be scaled by `rayIntensity` since the ray may have ricocheted many times before it arrived at that matte surface in the first place. Putting all of these multipliers together, we arrive at the code where `CalculateLighting` calls `CalculateMatte`:

```
if (opacity > 0.0)
{
    // This object is at least a little bit opaque,
    // so calculate the part of the color caused by
    // matte (scattered) reflection.
    colorSum +=
        opacity *
        optics.GetMatteColor() *
        rayIntensity *
        CalculateMatte(intersection);
}
```

Chapter 9

Refraction

9.1 Why refraction before reflection?

Usually I like to explain simpler things before more complicated things, but this is an exception to that rule. Refraction is more complicated than mirror reflection, both mathematically and algorithmically, but I discuss it first because we cannot calculate mirror reflection without calculating refraction first. This is because we don't know how much light reflects from a transparent (or partially transparent) object until we know what portion of light refracts through it. As mentioned briefly before, there are two physical effects that add up to the total mirror reflection: refractive reflection and glossy reflection. A transparent substance causes a ray of light to split into two weaker rays, one that is bent and passes through the substance and another that is reflected from the surface. Once we determine what portion of the light ray is reflected due to refraction, we can combine that with the effects of glossy reflection to calculate total mirror reflection.

9.2 Understanding the physics of refraction

But first we will focus on refraction. When a ray of light passes from one transparent substance into another one, it generally changes its direction, deviating from the straight line path it was on. The amount by which it bends depends on the angle of the ray of light with respect to the surface, along with the optical nature of the two substances involved. Interestingly, the change in direction results because of changes in the speed of light passing through different substances, and the fact that light behaves like waves. Light travels through a vacuum at about 3×10^8 meters per second, but it slows down significantly when passing through transparent matter like water or glass.

When light passes at an angle from one substance into another that causes it to travel slower, the parts of a given light wave that reach the surface first are slowed down before other parts of the same wave that reach the surface a little later. As seen in Figure 9.1, because different parts of the same wave are slowed down at different times, the direction of the wave is bent. The new direction is closer to being perpendicular to the surface.

To help understand why the speed change causes a direction change, imagine you are pulling a cart in a straight line across a concrete road. Suppose your path wanders off the left side of the road into some soft sand. For the period of time that the left wheels are slowed down by the sand, but the right wheels are still on concrete, the cart would inevitably start to curve more toward the left. If a little later the right wheels enter the sand also and are forced to slow down to the same speed as the left wheels, the cart would then tend to move along a straight path again, but more slowly, and on a different heading than when you started.

The same figure shows how refraction in the opposite direction occurs when the wave emerges from the other side of the substance: the parts of a wave that exit the substance first speed up before other parts, causing the wave to bend away from the surface's perpendicular line.

If c represents the speed of light in a vacuum and v represents the speed of light in a transparent substance, we can use the ratio $N = \frac{c}{v}$ to calculate how much that substance causes light to bend under various circumstances. We will call N the *refractive index* of the substance. Because the refractive index is the ratio of two speeds, no matter what speed units are used to express them (so long as both speeds are measured using the same units: meters per second, miles per

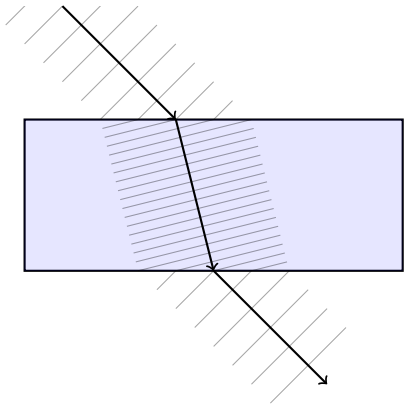

Figure 9.1: Waves of light entering a substance at some angle from the perpendicular. The substance slows down the light waves, causing them to be spaced closer together. The left part of each wave enters the substance before the right part, causing it to slow down first. This forces the entire wave to change direction. The inverse happens when the waves leave the substance.

hour, etc.), the units cancel out by division and the refractive index has same dimensionless numeric value. The refractive index N is inversely related to the speed of light v passing through the substance. The more the substance slows down light, the more v gets smaller and therefore the larger N gets. Nothing can make light travel faster than c (at least according to Einstein's theories of relativity), so actual substances in the real world always have $N \geq 1$. Typical values for N are 1.333 for water and 1.5 to 1.6 for common kinds of glass. Toward the lower end of the scale is air at 1.0003. One of the higher values for refractive index is diamond, having $N = 2.419$. The C++ code that accompanies this book allows a wide range of refractive index values, and the header file `imager.h` lists several common values for convenience:

```
const double REFRACTION_VACUUM   = 1.0000;
const double REFRACTION_AIR      = 1.0003;
const double REFRACTION_ICE      = 1.3100;
const double REFRACTION_WATER    = 1.3330;
const double REFRACTION_GASOLINE = 1.3980;
const double REFRACTION_GLASS    = 1.5500;
const double REFRACTION_SAPPHIRE = 1.7700;
const double REFRACTION_DIAMOND  = 2.4190;
```

Wikipedia has a more complete list of refractive indices for the curious reader:

`http://en.wikipedia.org/wiki/List_of_refractive_indices`

9.3 Snell's Law adapted for vectors

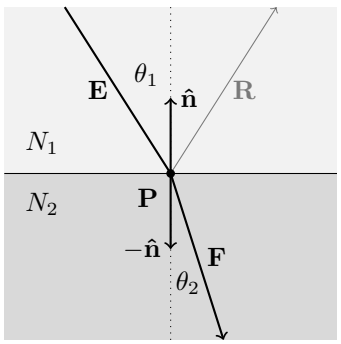

Figure 9.2: Light passing into a substance with a **higher** refractive index ($N_2 > N_1$) is bent **toward** a line perpendicular to the surface. The light ray **E** passes through the top substance (whose refractive index is N_1), strikes the boundary between the two substances at intersection point **P**, and most of it continues through the bottom substance (whose refractive index is N_2) along the direction vector **F**. A small portion of the light is reflected along **R**.

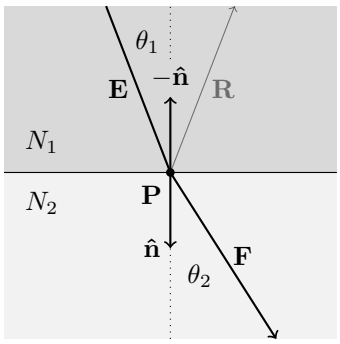

Figure 9.3: Light passing into a substance with a **lower** refractive index ($N_2 < N_1$) is bent **away** from a line perpendicular to the surface. As in the previous figure, the light travels from the top along **E**, strikes the intersection point **P**, and bends to travel in a new direction **F**. (As before, a small amount reflects in the direction **R**).

9.3.1 Introduction to Snell's Law

Figure 9.2 shows how a ray of light bends when it passes from one substance into another having a higher refractive index, and Figure 9.3 shows the inverse case, where a ray of light passes from one substance into another with a lower refractive index. In the first case, the ray is bent to a smaller angle from a line perpendicular to the surface, and in the second case, the ray is bent to a larger angle. Both cases can be treated as mathematically identical, using the following emperical equation known as *Snell's Law*:

$$N_1 \sin(\theta_1) = N_2 \sin(\theta_2)$$

In both figures, N_1 is the refractive index of the substance on the top and N_2 is that of the bottom substance. Both figures show a light ray **E** passing from the top substance into the bottom substance. The light ray strikes the boundary point **P** between the two substances at an angle θ_1 from the dotted perpendicular line, and is bent to a new angle θ_2 from the perpendicular, thus traveling along a new direction **F**.

9.3.2 Refractive reflection

As anyone knows who has looked at a pane of glass or a puddle of clear water, transparent substances form mirror-like images. Physicists determined long ago that reflection and refraction are inseparable. Any substance that refracts light will also reflect it. An interface between transparent substances effectively splits the light energy along two rays, one that reflects back into the substance from which the incident light originated and the other that refracts into the substance on the other side of the interface. The amount of incident light that the surface reflects depends on the angle that the light strikes that surface. There is always some amount of reflection, but the amount increases nonlinearly the more an incident ray slants from a line perpendicular with the surface.

9.3.3 Special case: total internal reflection

When a light ray passes from one substance into another of lower refractive index, for example from water into air, there is a limit to how steep the entry angle θ_1 can be before refraction is no longer possible. When θ_1 increases to a value such that θ_2 reaches 90°, refraction ceases and all of the light energy is reflected. The value of θ_1 where this occurs is called the *critical angle of incidence*, which we will designate θ_c. Figure 9.4 shows the same optical setup as 9.3, only with the incident light ray **E** at the critical angle θ_c. We can use Snell's Law to calculate θ_c by replacing θ_1 with θ_c and setting $\theta_2 = 90°$, which means that $\sin(\theta_2) = 1$. Solving for θ_c gives us:

$$\theta_c = \arcsin\left(\frac{N_2}{N_1}\right)$$

The "arcsin" here is the inverse sine, a function that returns the angle whose sine is its argument. Because the sine of an angle must be somewhere between -1 and $+1$, the inverse sine

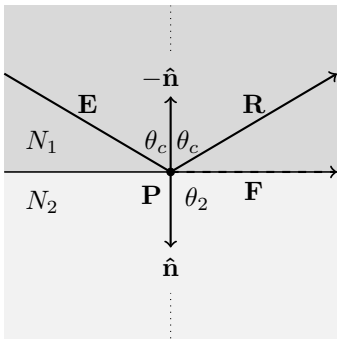

Figure 9.4: A ray of light **E** passing into a substance of lower refractive index at the **critical angle of incidence** θ_c. Snell's Law indicates that refracted ray **F** is bent to $\theta_2 = 90^\circ$ from the dotted perpendicular line, meaning it grazes the boundary between the two surfaces. In practice, at the critical angle and beyond, refraction vanishes and all of the light is reflected along the direction **R**.

is defined only when its argument is between -1 and $+1$. If $N_2 > N_1$, the arcsin argument goes out of bounds, and therefore there is no critical angle θ_c. This is why total reflection can occur only when the light travels into a substance with a lower refractive index.

9.3.4 Setting up the vector equations for refraction

If our task were simply the calculation of the angle θ_2, things would be quite simple. Unfortunately, we are dealing with vectors in three-dimensional space. We know the two refractive indices N_1 and N_2, the incident light ray vector **E**, and the surface normal vector $\hat{\mathbf{n}}$ indicating the direction perpendicular to the interface between the two substances. From these known quantities, we need to calculate the direction vector **F** of the refracted light ray. To set up the solution, we first define unit vectors that point in the same direction as **E** and **F**:

$$\hat{\mathbf{e}} = \frac{\mathbf{E}}{|\mathbf{E}|} \qquad \hat{\mathbf{f}} = \frac{\mathbf{F}}{|\mathbf{F}|}$$

Using the fact that the dot product of two unit vectors is equal to the cosine of their included angle, in combination with Snell's Law, we can write the following system of equations that indirectly relate the refractive indices N_1 and N_2 with the unit vectors $\hat{\mathbf{e}}$ and $\hat{\mathbf{f}}$ via the incident angle θ_1 and the refraction angle θ_2.

$$\begin{cases} N_1 \sin(\theta_1) = N_2 \sin(\theta_2) \\ \hat{\mathbf{e}} \cdot \hat{\mathbf{n}} = \pm \cos(\theta_1) \\ \hat{\mathbf{f}} \cdot \hat{\mathbf{n}} = \pm \cos(\theta_2) \end{cases}$$

9.3.5 Folding the three equations into one

The $\pm$ symbols in the second and third equation result from the ambiguity of the normal vector $\hat{\mathbf{n}}$: a careful consideration reveals that there is no relationship between the ray entering or leaving a particular object and whether the refractive index is increasing or decreasing. The incident ray unit vector $\hat{\mathbf{e}}$ might be going "against" the normal vector $\hat{\mathbf{n}}$ or "with" it, and in either case the light ray may be entering a region of higher or lower refractive index. The two concepts are completely independent. The ambient refractive index may be higher or lower than the refractive index inside the object, the ray may be entering or leaving the object, and furthermore, the ray may be passing into an object that is embedded inside another object.

In mathematical terms, we want to describe the angle θ_1 as the smallest angle between the vector **E** (or its unit vector counterpart $\hat{\mathbf{e}}$) and the dotted perpendicular line, whether $\hat{\mathbf{n}}$ is on the same side of the surface as the incident light ray **E** (as shown in Figure 9.2) or on the opposite side (as shown in Figure 9.3). In the first case, we would have $\hat{\mathbf{e}} \cdot \hat{\mathbf{n}} = \cos(\theta_1)$, and in the second case $\hat{\mathbf{e}} \cdot (-\hat{\mathbf{n}}) = -\hat{\mathbf{e}} \cdot \hat{\mathbf{n}} = \cos(\theta_1)$. Putting both cases together results in $\hat{\mathbf{e}} \cdot \hat{\mathbf{n}} = \pm \cos(\theta_1)$, as

shown in the system of equations above. The same reasoning applies to the $\pm$ that appears in the equation for $\hat{\mathbf{f}}$.

Yet we need to use the normal vector as part of the mathematical solution, because when we find an intersection, it is the only thing available to us that describes the orientation of the boundary between the two substances. There is a happy way to circumvent this problem, however: we square both sides of all three of the preceeding equations, eliminating the both instances of $\pm$.

$$\begin{cases} N_1^2 \sin^2(\theta_1) = N_2^2 \sin^2(\theta_2) \\ (\hat{\mathbf{e}} \cdot \hat{\mathbf{n}})^2 = \cos^2(\theta_1) \\ (\hat{\mathbf{f}} \cdot \hat{\mathbf{n}})^2 = \cos^2(\theta_2) \end{cases}$$

A quick note on notation: when you see $\sin^2(x)$ or $\cos^2(x)$, it means the same thing as $(\sin(x))^2$ or $(\cos(x))^2$, namely taking the sine or cosine of the angle and multiplying the result by itself.

Squaring the equations also lets us further the solution by use of a trigonometric identity that relates the sine and cosine of any angle x:

$$\cos^2(x) + \sin^2(x) = 1$$

We can thus express the squared sines in terms of the squared cosines by rearranging the trig identity as:

$$\sin^2(x) = 1 - \cos^2(x)$$

This lets us replace the squared sine terms in the first equation with squared cosine terms:

$$\begin{cases} N_1^2 \left(1 - \cos^2(\theta_1)\right) = N_2^2 \left(1 - \cos^2(\theta_2)\right) \\ (\hat{\mathbf{e}} \cdot \hat{\mathbf{n}})^2 = \cos^2(\theta_1) \\ (\hat{\mathbf{f}} \cdot \hat{\mathbf{n}})^2 = \cos^2(\theta_2) \end{cases}$$

By substituting the second and third equations into the first, we completely eliminate the angles and trig functions to obtain a single equation:

$$N_1^2\Big(1 - (\hat{\mathbf{e}} \cdot \hat{\mathbf{n}})^2\Big) = N_2^2\Big(1 - (\hat{\mathbf{f}} \cdot \hat{\mathbf{n}})^2\Big)$$

9.3.6 Dealing with the double cone

We could solve this equation for the unknown value $(\hat{\mathbf{f}} \cdot \hat{\mathbf{n}})^2$ in terms of the other quantities (which all have known values), but we would still run into a problem: there are an infinite number of unit vectors $\hat{\mathbf{f}}$ whose squared dot product with the normal vector $\hat{\mathbf{n}}$ would yield the known numeric quantity on the other side of the equation. Although $\hat{\mathbf{n}}$ is a known vector, and thus its position is fixed in space, there are an infinite number of unit vectors $\hat{\mathbf{f}}$ whose squared dot product with $\hat{\mathbf{n}}$ will be equal to the value $\cos^2(\theta_2)$. Figure 9.5 shows that the set of all such $\hat{\mathbf{f}}$ sweeps out a pair of cone shapes from the incidence point $\mathbf{P}$, one cone for each side of the surface.

Our original goal was to find the direction vector $\mathbf{F}$ or its unit vector counterpart $\hat{\mathbf{f}}$. (We aren't picky here; any vector pointing in the same direction will be fine, meaning any vector of the form $\mathbf{F} = p\hat{\mathbf{f}}$ where p is a positive scalar.) Clearly we need something extra to further constrain the solution to provide a specific answer for $\mathbf{F}$ itself. Our first clue is that the vectors $\mathbf{E}$, $\hat{\mathbf{n}}$, and $\mathbf{F}$ must all lie in the same plane, because to do otherwise would mean that the ray of light would be bending in an asymmetric way. Imagine yourself inside the top substance looking down at the point $\mathbf{P}$ exactly perpendicular to the boundary. Imagine also that the light ray $\mathbf{E}$ is descending onto the surface from the north. Both of the involved materials are assumed to be completely uniform in composition, so there is no reason to think they would make the ray veer even slightly to the east or west; why would one of those directions be preferred over the other? No, such a ray of light must continue toward the south, perhaps descending at a steeper or shallower angle depending on the refractive indices, but due south nonetheless.

Constraining $\mathbf{F}$ (and therefore $\hat{\mathbf{f}}$) to lie in the same plane as $\hat{\mathbf{n}}$ and $\mathbf{E}$ leads to a finite solution set for $\hat{\mathbf{f}}$. The double cone intersects with this plane such that there are now only four possibile directions for $\hat{\mathbf{f}}$, as shown in Figure 9.6. Only one of these four directions is the correct solution

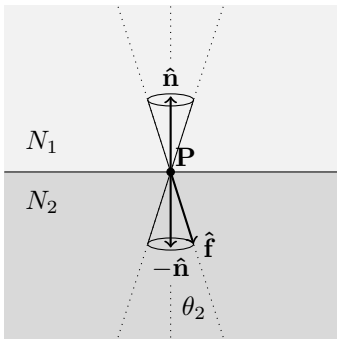

Figure 9.5: For specific values of $\hat{\mathbf{n}}$ and θ_2, the set of all $\hat{\mathbf{f}}$ such that $(\hat{\mathbf{f}} \cdot \hat{\mathbf{n}})^2 = \cos^2(\theta_2)$ sweeps out an infinite number of directions from the intersection point $\mathbf{P}$, forming a double-cone shape. We want to solve for the actual refraction vector $\hat{\mathbf{f}}$ as labeled in the figure.

for $\hat{\mathbf{f}}$. The other three are "phantom" solutions that we must eliminate somehow. But we are making progress!

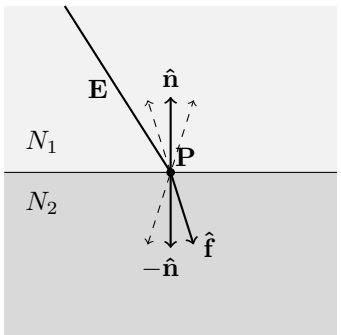

Figure 9.6: Requiring $\hat{\mathbf{f}}$ to lie in the same plane as $\mathbf{E}$ and $\hat{\mathbf{n}}$ can be visualized as slicing that common plane through the double cones from the previous figure. This results in a set of four possible solutions for $\hat{\mathbf{f}}$ instead of an infinite set. We still need to figure out how to pick the correct $\hat{\mathbf{f}}$ while discarding the three "phantom" solutions, shown here as dashed vectors.

9.3.7 Constraining F to the correct plane

But before we get ahead of ourselves, how do we write another equation that mathematically constrains $\mathbf{F}$ to the same plane as $\mathbf{E}$ and $\hat{\mathbf{n}}$ in the first place? The answer is to write $\mathbf{F}$ as a linear combination of the incidence unit vector $\hat{\mathbf{e}}$ and the surface normal unit vector $\hat{\mathbf{n}}$:

$$\mathbf{F} = \hat{\mathbf{e}} + k\hat{\mathbf{n}}$$

Here, k is a scalar parameter that determines how far the light ray is bent from its original direction $\hat{\mathbf{e}}$, either toward or away from the normal vector $\hat{\mathbf{n}}$. Depending on the situation, the as yet unknown value of k may be negative, positive, or zero.

Recalling that $\hat{\mathbf{f}} = \frac{\mathbf{F}}{|\mathbf{F}|}$, we can now write $\hat{\mathbf{f}}$ as

$$\hat{\mathbf{f}} = \frac{\hat{\mathbf{e}} + k\hat{\mathbf{n}}}{|\hat{\mathbf{e}} + k\hat{\mathbf{n}}|}$$

Now we substitute this rewritten form of $\hat{\mathbf{f}}$ into the earlier equation where we eliminated the squared sines and cosines:

$$N_1^2\Big(1 - (\hat{\mathbf{e}} \cdot \hat{\mathbf{n}})^2\Big) = N_2^2 \left(1 - \left(\frac{\hat{\mathbf{e}} + k\hat{\mathbf{n}}}{|\hat{\mathbf{e}} + k\hat{\mathbf{n}}|} \cdot \hat{\mathbf{n}}\right)^2\right)$$

9.3.8 Reducing to a scalar equation in k

At this point we have a single equation with a single unknown k. Solving for k is a little bit of work. We start by expanding the squared dot product term on the right side of the equation. Let's focus on that part alone for a little while. Later we will take the result and plug it back into the previous equation. First we factor out the magnitude term $|\hat{\mathbf{e}}+k\hat{\mathbf{n}}|$ in the denominator. We can do this because it is a scalar.

$$\left(\frac{\hat{\mathbf{e}}+k\hat{\mathbf{n}}}{|\hat{\mathbf{e}}+k\hat{\mathbf{n}}|}\cdot\hat{\mathbf{n}}\right)^2=\frac{\left((\hat{\mathbf{e}}+k\hat{\mathbf{n}})\cdot\hat{\mathbf{n}}\right)^2}{|\hat{\mathbf{e}}+k\hat{\mathbf{n}}|^2}$$

Now we distribute the dot product in the numerator in a very similar way to distributing a scalar sum multiplied by another scalar in familiar algebra. This is possible because

$$\begin{aligned}
&(\hat{\mathbf{e}}+k\hat{\mathbf{n}})\cdot\hat{\mathbf{n}}\\
=&\Big((e_x,\,e_y,\,e_z)+k(n_x,\,n_y,\,n_z)\Big)\cdot(n_x,\,n_y,\,n_z)\\
=&(e_x+kn_x,\,e_y+kn_y,\,e_z+kn_z)\cdot(n_x,\,n_y,\,n_z)\\
=&((e_x+kn_x)n_x,\,(e_y+kn_y)n_y,\,(e_z+kn_z)n_z)\\
=&(e_xn_x+kn_x^2,\,e_yn_y+kn_y^2,\,e_zn_z+kn_z^2)\\
=&(e_xn_x,\,e_yn_y,\,e_zn_z)+k(n_x^2,\,n_y^2,\,n_z^2)\\
=&(\hat{\mathbf{e}}\cdot\hat{\mathbf{n}})+k(\hat{\mathbf{n}}\cdot\hat{\mathbf{n}})
\end{aligned}$$

If you look at the first line and last line of this derivation, it looks just like what you would expect if you were doing algebra with ordinary multiplication instead of vector dot products. We will use this dot product distribution trick again soon, so it is important to understand.

We can simplify the expression further by noting that the dot product of any unit vector by itself is equal to 1. As with any vector, $\hat{\mathbf{n}}\cdot\hat{\mathbf{n}}=|\hat{\mathbf{n}}|^2$, but because $\hat{\mathbf{n}}$ is a unit vector, $|\hat{\mathbf{n}}|^2=1$, by definition. Substituting $\hat{\mathbf{n}}\cdot\hat{\mathbf{n}}=1$, we find that

$$(\hat{\mathbf{e}}+k\hat{\mathbf{n}})\cdot\hat{\mathbf{n}}=(\hat{\mathbf{e}}\cdot\hat{\mathbf{n}})+k$$

Now let's do some work on the squared magnitude term $|\hat{\mathbf{e}}+k\hat{\mathbf{n}}|^2$ we factored out earlier. We take advantage of the fact that the square of any vector's magnitude is the same as the dot product of that vector with itself: for any vector $\mathbf{A}$, we know that $|\mathbf{A}|^2=\mathbf{A}\cdot\mathbf{A}=A_x^2+A_y^2+A_z^2$. Applying this concept to $|\hat{\mathbf{e}}+k\hat{\mathbf{n}}|^2$, and noting again that $\hat{\mathbf{e}}\cdot\hat{\mathbf{e}}=1$ and $\hat{\mathbf{n}}\cdot\hat{\mathbf{n}}=1$, we have

$$\begin{aligned}
&|\hat{\mathbf{e}}+k\hat{\mathbf{n}}|^2\\
=&(\hat{\mathbf{e}}+k\hat{\mathbf{n}})\cdot(\hat{\mathbf{e}}+k\hat{\mathbf{n}})\\
=&(\hat{\mathbf{e}}\cdot\hat{\mathbf{e}})+2k(\hat{\mathbf{e}}\cdot\hat{\mathbf{n}})+k^2(\hat{\mathbf{n}}\cdot\hat{\mathbf{n}})\\
=&1+2k(\hat{\mathbf{e}}\cdot\hat{\mathbf{n}})+k^2
\end{aligned}$$

This means the term we were working on becomes

$$\left(\frac{\hat{\mathbf{e}}+k\hat{\mathbf{n}}}{|\hat{\mathbf{e}}+k\hat{\mathbf{n}}|}\cdot\hat{\mathbf{n}}\right)^2=\frac{\left((\hat{\mathbf{e}}\cdot\hat{\mathbf{n}})+k\right)^2}{1+2k(\hat{\mathbf{e}}\cdot\hat{\mathbf{n}})+k^2}=\frac{(\hat{\mathbf{e}}\cdot\hat{\mathbf{n}})^2+2k(\hat{\mathbf{e}}\cdot\hat{\mathbf{n}})+k^2}{1+2k(\hat{\mathbf{e}}\cdot\hat{\mathbf{n}})+k^2}$$

This looks a little intimidating, but things become much more pleasant if you note that since both $\hat{\mathbf{e}}$ and $\hat{\mathbf{n}}$ are known vectors, their dot product is a fixed scalar value. Let's make up a new variable α just to make the math easier to follow:

$$\alpha=\hat{\mathbf{e}}\cdot\hat{\mathbf{n}}$$

The previous equation then becomes

$$\left(\frac{\hat{\mathbf{e}}+k\hat{\mathbf{n}}}{|\hat{\mathbf{e}}+k\hat{\mathbf{n}}|}\cdot\hat{\mathbf{n}}\right)^2=\frac{\alpha^2+2\alpha k+k^2}{1+2\alpha k+k^2}$$

We now return to the equation we derived from Snell's Law:

$$N_1^2\Big(1-(\hat{\mathbf{e}}\cdot\hat{\mathbf{n}})^2\Big) = N_2^2\left(1-\left(\frac{\hat{\mathbf{e}}+k\hat{\mathbf{n}}}{|\hat{\mathbf{e}}+k\hat{\mathbf{n}}|}\cdot\hat{\mathbf{n}}\right)^2\right)$$

Using the α substitution and the algebraic work we just did, this equation transforms into a more managable equation involving only scalars.

$$N_1^2(1-\alpha^2) = N_2^2\left(1-\frac{\alpha^2+2\alpha k+k^2}{1+2\alpha k+k^2}\right)$$

$$\left(\frac{N_1}{N_2}\right)^2(1-\alpha^2) = \frac{(1+2\alpha k+k^2)-(\alpha^2+2\alpha k+k^2)}{1+2\alpha k+k^2}$$

$$\left(\frac{N_1}{N_2}\right)^2(1-\alpha^2) = \frac{1-\alpha^2}{1+2\alpha k+k^2}$$

$$1+2\alpha k+k^2 = \left(\frac{N_2}{N_1}\right)^2$$

Desiring to solve for k, we write the equation in standard quadratic form:

$$k^2+2\alpha k+\left(1-\left(\frac{N_2}{N_1}\right)^2\right) = 0$$

9.3.9 Picking the correct solution for k and F

The C++ code for `CalculateRefraction` solves the quadratic equation for k using the helper function `SolveQuadraticEquation` from `algebra.h`, just as the `Sphere` class did for finding intersections with a sphere. As is typical for quadratic equations, there are generally two solutions for k. Only one of the two values of k yields a correct value of $\mathbf{F} = \hat{\mathbf{e}} + k\hat{\mathbf{n}}$, while the other produces one of the phantom vectors as depicted in Figure 9.6.

The C++ code figures out which value of $\mathbf{F}$ is correct by determining which one is most aligned with the incident ray $\mathbf{E}$. As you can see in Figure 9.6, the correct value of $\mathbf{F}$ causes the path of the light ray to bend less than any of the three phantom vectors. This means that the correct value of k leads to a value of $\mathbf{E}\cdot\mathbf{F}$ greater than for the phantom vector produced by using the incorrect k value. The code tries both values of k, calculates $\mathbf{F} = \hat{\mathbf{e}} + k\hat{\mathbf{n}}$ for both values, and picks the k that maximizes the dot product $\mathbf{E}\cdot\mathbf{F}$. Once the correct value of $\mathbf{F}$ is found, it tells us which way a refracted ray of light travels after passing through the boundary between the two substances.

9.4 Calculating refractive reflection

Up to this point we have been concentrating on calculating the refracted ray while ignoring the part of the light energy that goes into reflection. The member function `CalculateRefraction` does **not** need to figure out the direction vector of the refracted ray $\mathbf{R}$. That will be the job of another member function `CalculateReflection`, which is the topic of the next chapter. However, `CalculateRefraction` **does** need to figure out what scalar portion of the incident light is reflected due to refraction. To fulfill this part of its contract, `CalculateRefraction` is responsible for assigning a value between 0 and 1 to its output parameter `outReflectionFactor`. The reflection factor is closer to 0 when the incidence angle is nearly perpendicular to the surface, and increases to 1 when the incidence angle reaches or exceeds the critical angle θ_c.

I will not go into much detail about how to derive the formulas for calculating the reflection factor. I will mention that they are called the *Fresnel equations*, and the curious reader can find out more about them in the following Wikipedia article.

http://en.wikipedia.org/wiki/Fresnel_equations

The Fresnel equations give separate answers depending on the polarization of the incident light. This C++ code does not model light polarization. Instead, it calculates the reflection factor by averaging the higher and lower extremes predicted by the Fresnel equations. It finds the higher and lower values from two consecutive calls to the helper function `PolarizedReflection`, and assigns the average of the resulting return values to the output parameter `outReflectionFactor`.

Keep in mind that the complexities of finding the refraction vector and the reflection factor disappear when `CalculateRefraction` detects that the incident ray experiences total internal reflection. In that case, the function exits early after setting `outReflectionFactor` to 1, while returning a pure black color value:

```
if (sin_a2 <= -1.0 || sin_a2 >= +1.0)
{
    // Since sin_a2 is outside the bounds -1..+1, then
    // there is no such real angle a2, which in turn
    // means that the ray experiences total internal reflection,
    // so that no refracted ray exists.
    outReflectionFactor = 1.0;      // complete reflection
    return Color(0.0, 0.0, 0.0);    // no refraction at all
}
```

9.5 Ambient refractive index

By default, the ray tracing code presented here treats any unoccupied region of space as having a vacuum's refractive index (exactly 1.0), but it is possible to override this default to create images that simulate other ambient environments, such as an underwater scene. To do this, call `Scene::SetAmbientRefraction` with the desired refractive index as its argument. Here is an example of the underwater simulation:

```
Scene scene;
scene.SetAmbientRefraction(REFRACTION_WATER);
```

In such an underwater scene, glass objects would become much more subtle, and spherical bubbles of air would become profoundly refractive, acting as lenses that make everything seen through them look tiny.

9.6 Determining a point's refractive index

A caller of `CalculateRefraction` must pass it the refractive index for the region of space that the ray of light was traversing before hitting the boundary between the two substances. This is the parameter `sourceRefractiveIndex`. However, it is the job of `CalculateRefraction` to figure out what the refractive index is on the other side of the boundary. It does this by extrapolating the line along the direction of the light ray a small distance beyond the intersection point to arrive at a test point. Then it must figure out which `SolidObject` instance (if any) contains that test point.

We encounter a dilemma if there is more than one solid that contains the test point. There is nothing in the ray tracer code that prevents two or more solids from overlapping. In fact, preventing overlap would be an undesirable limitation. What if we want to depict a jar of water with some glass marbles and air bubbles in it? In real life, the marbles and bubbles would be mutually exclusive with the water: wherever there is glass or air, there is no water. But in the ray tracer code, it is far simpler to define a cylindrical volume of water and then add glass spheres for the marbles and spheres of air for the bubbles that occupy the same region of space as the water. In such a scene, if a particular ray of light is passing through the water and strikes an air bubble, we want the air bubble to have priority over the water for determining the new refractive index.

In this same scene, it is conceivable that we could want to have a tiny drop of water inside one of the air bubbles. Here is a case where there is a small body of water overlapping with a larger body of air and an even larger body of surrounding water! We have three subtances overlapping. How is the code to know which substance controls the refractive index at a given

point in space? We can't just say that the lower (or higher) refractive index wins, because it would prevent this scene from looking right. It turns out that any attempt for the code to guess which object is dominant will eventually fail to make the programmer happy in some case. Only the programmer knows the artistic intent of which objects have priority over others, so the programmer must choose somehow.

The ray tracer uses a simple and arbitrary rule to break ties for the ownership of a point in space: whichever instance of `SolidObject` was added to the scene first has priority. It is up to you to call `Scene::AddSolidObject` in whatever order makes sense for the image you are trying to create. In the example here, you would add the bubbles and marbles to the scene before adding the cylindrical body of water. If you were to add the water first, you would still be able to see the marbles if they were a different color than the water, but the marbles would not bend rays of light, so they would look like colored regions of water instead of glass.

To find the owner of the test point, `CalculateRefraction` calls the member function `PrimaryContainer`, passing the test point as its argument. `PrimaryContainer` iterates through all of the objects in the scene in the same order they were added to the scene, calling the `Contains` method on each object in turn. The first object whose `Contains` method returns `true` for the test point is declared the owner of that point, and `PrimaryContainer` returns a pointer to that object. If none of the objects contain the test point, `PrimaryContainer` returns `NULL`. Therefore, `CalculateReflection` must check the returned pointer for being `NULL`. In that case, it uses the scene's ambient refractive index. If the returned pointer is not `NULL`, `CalculateRefraction` calls the containing object's `GetRefractiveIndex` method to determine the refractive index.

9.7 `CalculateRefraction` source code

Here is the complete code for `CalculateRefraction`, followed by the helper functions `PolarizedReflection` and `PrimaryContainer`.

```
Color Scene::CalculateRefraction(
    const Intersection& intersection,
    const Vector& direction,
    double sourceRefractiveIndex,
    Color rayIntensity,
    int recursionDepth,
    double& outReflectionFactor) const
{
    // Convert direction to a unit vector so that
    // relation between angle and dot product is simpler.
    const Vector dirUnit = direction.UnitVector();

    double cos_a1 = DotProduct(dirUnit, intersection.surfaceNormal);
    double sin_a1;
    if (cos_a1 <= -1.0)
    {
        if (cos_a1 < -1.0001)
        {
            throw ImagerException("Dot product too small.");
        }
        // The incident ray points in exactly the opposite
        // direction as the normal vector, so the ray
        // is entering the solid exactly perpendicular
        // to the surface at the intersection point.
        cos_a1 = -1.0;  // clamp to lower limit
        sin_a1 =  0.0;
    }
    else if (cos_a1 >= +1.0)
    {
        if (cos_a1 > +1.0001)
        {
            throw ImagerException("Dot product too large.");
        }
        // The incident ray points in exactly the same
        // direction as the normal vector, so the ray
        // is exiting the solid exactly perpendicular
        // to the surface at the intersection point.
        cos_a1 = +1.0;  // clamp to upper limit
        sin_a1 =  0.0;
```

```
}
else
{
    // The ray is entering/exiting the solid at some
    // positive angle with respect to the normal vector.
    // We need to calculate the sine of that angle
    // using the trig identity cos^2 + sin^2 = 1.
    // The angle between any two vectors is always between
    // 0 and PI, so the sine of such an angle is never negative.
    sin_a1 = sqrt(1.0 - cos_a1*cos_a1);
}

// The parameter sourceRefractiveIndex passed to this function
// tells us the refractive index of the medium the light ray
// was passing through before striking this intersection.
// We need to figure out what the target refractive index is,
// i.e., the refractive index of whatever substance the ray
// is about to pass into.  We determine this by pretending that
// the ray continues traveling in the same direction a tiny
// amount beyond the intersection point, then asking which
// solid object (if any) contains that test point.
// Ties are broken by insertion order: whichever solid was
// inserted into the scene first that contains a point is
// considered the winner.  If a solid is found, its refractive
// index is used as the target refractive index; otherwise,
// we use the scene's ambient refraction, which defaults to
// vacuum (but that can be overridden by a call to
// Scene::SetAmbientRefraction).

const double SMALL_SHIFT = 0.001;
const Vector testPoint = intersection.point + SMALL_SHIFT*dirUnit;
const SolidObject* container = PrimaryContainer(testPoint);
const double targetRefractiveIndex =
    (container != NULL) ?
    container->GetRefractiveIndex() :
    ambientRefraction;

const double ratio = sourceRefractiveIndex / targetRefractiveIndex;

// Snell's Law: the sine of the refracted ray's angle
// with the normal is obtained by multiplying the
// ratio of refractive indices by the sine of the
// incident ray's angle with the normal.
const double sin_a2 = ratio * sin_a1;

if (sin_a2 <= -1.0 || sin_a2 >= +1.0)
{
    // Since sin_a2 is outside the bounds -1..+1, then
    // there is no such real angle a2, which in turn
    // means that the ray experiences total internal reflection,
    // so that no refracted ray exists.
    outReflectionFactor = 1.0;      // complete reflection
    return Color(0.0, 0.0, 0.0);    // no refraction at all
}

// Getting here means there is at least a little bit of
// refracted light in addition to reflected light.
// Determine the direction of the refracted light.
// We solve a quadratic equation to help us calculate
// the vector direction of the refracted ray.

double k[2];
const int numSolutions = Algebra::SolveQuadraticEquation(
    1.0,
    2.0 * cos_a1,
    1.0 - 1.0/(ratio*ratio),
    k);

// There are generally 2 solutions for k, but only
// one of them is correct.  The right answer is the
// value of k that causes the light ray to bend the
// smallest angle when comparing the direction of the
// refracted ray to the incident ray.  This is the
```

```
    // same as finding the hypothetical refracted ray
    // with the largest positive dot product.
    // In real refraction, the ray is always bent by less
    // than 90 degrees, so all valid dot products are
    // positive numbers.
    double maxAlignment = -0.0001;  // any negative number works as a flag
    Vector refractDir;
    for (int i=0; i < numSolutions; ++i)
    {
        Vector refractAttempt = dirUnit + k[i]*intersection.surfaceNormal;
        double alignment = DotProduct(dirUnit, refractAttempt);
        if (alignment > maxAlignment)
        {
            maxAlignment = alignment;
            refractDir = refractAttempt;
        }
    }

    if (maxAlignment <= 0.0)
    {
        // Getting here means there is something wrong with the math.
        // Either there were no solutions to the quadratic equation,
        // or all solutions caused the refracted ray to bend 90 degrees
        // or more, which is not possible.
        throw ImagerException("Refraction failure.");
    }

    // Determine the cosine of the exit angle.
    double cos_a2 = sqrt(1.0 - sin_a2*sin_a2);
    if (cos_a1 < 0.0)
    {
        // Tricky bit: the polarity of cos_a2 must
        // match that of cos_a1.
        cos_a2 = -cos_a2;
    }

    // Determine what fraction of the light is
    // reflected at the interface.  The caller
    // needs to know this for calculating total
    // reflection, so it is saved in an output parameter.

    // We assume uniform polarization of light,
    // and therefore average the contributions of s-polarized
    // and p-polarized light.
    const double Rs = PolarizedReflection(
        sourceRefractiveIndex,
        targetRefractiveIndex,
        cos_a1,
        cos_a2);

    const double Rp = PolarizedReflection(
        sourceRefractiveIndex,
        targetRefractiveIndex,
        cos_a2,
        cos_a1);

    outReflectionFactor = (Rs + Rp) / 2.0;

    // Whatever fraction of the light is NOT reflected
    // goes into refraction.  The incoming ray intensity
    // is thus diminished by this fraction.
    const Color nextRayIntensity =
        (1.0 - outReflectionFactor) * rayIntensity;

    // Follow the ray in the new direction from the intersection point.
    return TraceRay(
        intersection.point,
        refractDir,
        targetRefractiveIndex,
        nextRayIntensity,
        recursionDepth);
}
```

```
double Scene::PolarizedReflection(
    double n1,              // source material's index of refraction
    double n2,              // target material's index of refraction
    double cos_a1,          // incident or outgoing ray angle cosine
    double cos_a2) const    // outgoing or incident ray angle cosine
{
    const double left  = n1 * cos_a1;
    const double right = n2 * cos_a2;
    double numer = left - right;
    double denom = left + right;
    denom *= denom;     // square the denominator
    if (denom < EPSILON)
    {
        // Assume complete reflection.
        return 1.0;
    }
    double reflection = (numer*numer) / denom;
    if (reflection > 1.0)
    {
        // Clamp to actual upper limit.
        return 1.0;
    }
    return reflection;
}

const SolidObject* Scene::PrimaryContainer(const Vector& point) const
{
    SolidObjectList::const_iterator iter = solidObjectList.begin();
    SolidObjectList::const_iterator end  = solidObjectList.end();
    for (; iter != end; ++iter)
    {
        const SolidObject* solid = *iter;
        if (solid->Contains(point))
        {
            return solid;
        }
    }

    return NULL;
}
```

Chapter 10

Mirror Reflection

10.1 The physics of mirror reflection

Like refraction, mirror reflection is a phenomenon that occurs at the boundary between two substances. In both cases, a portion of a light ray's energy is sent in a new direction when that ray hits a point on the boundary. Unlike refraction, mirror reflection causes light to bend back into the substance from which it came, instead of bending along a path into the new substance. Mirror reflection is also mathematically much simpler than refraction. The refractive indices of the two substances affect the amount of reflected light, but they have nothing to do with the reflected light's direction: light always reflects at the same angle as it arrived, as mesured from a line perpendicular to the surface. Just as we saw with refraction, the reflected ray lies in the same plane as both the incident ray and the perpendicular line, only on the opposite side of the perpendicular line as the incident ray. See Figure 10.1.

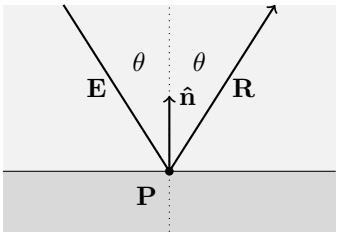

Figure 10.1: The incident ray **E** strikes the boundary between two substances at point **P**. The ray reflects in the direction **R**. The rays **E** and **R** both extend the same angle θ from the surface's perpendicular.

10.2 Two kinds of reflection, same vectors

In Chapter 7, we learned that this ray tracer source code models mirror reflection as the sum of two separate kinds of reflection: *glossy reflection* and *refractive reflection*. To recap, glossy reflection belongs to an imaginary coating on an object's surface. Refractive reflection, however, is determined by the refractive indices of substances on opposite sides of their common boundary. Glossy reflection may vary across an object's surface, but is equal in intensity regardless of the angle of the light's incidence; refractive reflection is more intense for light rays that arrive at a larger angle from the perpendicular. Glossy reflection can impart a color bias onto reflected light, but refractive reflection bounces red, green, and blue light equally.

In spite of all of these differences, they share an important geometrical trait. In both kinds of mirror reflection, the same incident ray **E** results in a reflected ray traveling in exactly the same outgoing direction **R**, as depicted in Figure 10.1. For this reason, the ray tracer code calculates the intensities and colors of glossy reflection and refractive reflection separately, but adds them together and calculates their common direction **R** using a single member function `CalculateReflection`.

10.3 Deriving the vector formula

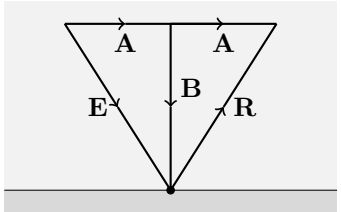

Figure 10.2: Vectors used in the derivation of the reflection formula.

We now consider the problem of calculating the mirror reflection vector $\mathbf{R}$, given the incident light ray vector $\mathbf{E}$ and the surface normal vector $\hat{\mathbf{n}}$. The first step is to break $\mathbf{E}$ into the sum of two vectors $\mathbf{A}$ and $\mathbf{B}$, such that $\mathbf{A}$ is parallel to the reflective surface and $\mathbf{B}$ is perpendicular to it, as depicted in Figure 10.2. Reflected light bounces away from the surface at the same angle as the incident light, so the angle between $\mathbf{E}$ and $\mathbf{B}$ is the same as the angle between $\mathbf{R}$ and $\mathbf{B}$. Two symmetric right triangles appear: $\mathbf{ABE}$ and $\mathbf{ABR}$.

We can see from Figure 10.2 that the following two vector equations must be true. (Initially, the second equation might be easier to understand rewritten as $\mathbf{B} + \mathbf{R} = \mathbf{A}$, but the form below is more useful to the upcoming discussion.)

$$\mathbf{E} = \mathbf{A} + \mathbf{B}$$
$$\mathbf{R} = \mathbf{A} - \mathbf{B}$$

We can find the perpendicular vector $\mathbf{B}$ by noting that since it is the component of $\mathbf{E}$ perpendicular to the surface, its magnitude must be equal to the dot product of $\mathbf{E}$ and the surface normal vector $\hat{\mathbf{n}}$. Likewise, $\mathbf{B}$ must point in either the same direction as $\hat{\mathbf{n}}$ or in the opposite direction $-\hat{\mathbf{n}}$. Regardless of the orientation of the $\hat{\mathbf{n}}$ (i.e., whether this reflection is occurring on the outside surface or inside surface of an instance of `SolidObject`), the following equation works.

$$\mathbf{B} = (\mathbf{E} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$$

To convince you that this equation works in both cases, consider each case separately. If $\hat{\mathbf{n}}$ points in the opposite direction as $\mathbf{B}$ (or upward as seen in Figure 10.2), then $\mathbf{E} \cdot \hat{\mathbf{n}}$ is a negative number, and multiplying any vector by a negative number results in another vector pointing in the opposite direction. Thus $(\mathbf{E} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$ points in the same direction as $\mathbf{B}$. Alternatively, if $\hat{\mathbf{n}}$ points in the same direction as $\mathbf{B}$ (downward as seen in the figure), the dot product is a positive number, and therefore $(\mathbf{E} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$ still points in the same direction as $\mathbf{B}$.

We now substitute the value of $\mathbf{B}$ into the equations for $\mathbf{E}$ and $\mathbf{R}$.

$$\mathbf{E} = \mathbf{A} + (\mathbf{E} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$$
$$\mathbf{R} = \mathbf{A} - (\mathbf{E} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$$

Subtracting the first equation from the second results in $\mathbf{A}$ canceling out, and adding $\mathbf{E}$ to both sides of this new equation results in the following formula for the reflected ray $\mathbf{R}$:

$$\mathbf{R} = \mathbf{E} - 2(\mathbf{E} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$$

This is exactly what we wanted. We now know how to calculate $\mathbf{R}$ using only $\mathbf{E}$ and $\hat{\mathbf{n}}$.

10.4 Recursive call to `CalculateLighting`

Once `CalculateReflection` uses the preceding formula to calculate the direction of the reflected ray, it must figure out where that ray goes in order to determine what effect the ray has on the

image. The intersection point where the ray reflected becomes a new vantage point, and the reflected ray direction **R** becomes the new direction to look.

`CalculateReflection` calls `TraceRay` to search in the direction **R** from the new vantage point **P**. If `TraceRay` finds an intersection in that direction, it calls `CalculateLighting` to complete the evaluation of the reflected ray. Because `CalculateLighting` and `CalculateReflection` call each other through the intermediate function `TraceRay`, their mutual recursion must be protected from excessive call depth. `TraceRay` thus adds 1 to whatever recursion depth is passed to it and sends that increased value to `CalculateLighting`. The latter stops the recursion if the maximum allowed depth has been reached.

As in any other case, if `TraceRay` finds no intersection in the specified direction, it knows that the ray continues infinitely into space and therefore returns the background color diminished by the ray intensity that was passed into it. Recall that the `rayIntensity` parameter tracks how weak the red, green, and blue color components of a ray of light have become as the ray bounces around the scene.

10.5 `CalculateReflection` source code

```
Color Scene::CalculateReflection(
    const Intersection& intersection,
    const Vector& incidentDir,
    double refractiveIndex,
    Color rayIntensity,
    int recursionDepth) const
{
    // Find the direction of the reflected ray based on the incident ray
    // direction and the surface normal vector.  The reflected ray has
    // the same angle with the normal vector as the incident ray, but
    // on the opposite side of the cone centered at the normal vector
    // that sweeps out the incident angle.
    const Vector& normal = intersection.surfaceNormal;
    const double perp = 2.0 * DotProduct(incidentDir, normal);
    const Vector reflectDir = incidentDir - (perp * normal);

    // Follow the ray in the new direction from the intersection point.
    return TraceRay(
        intersection.point,
        reflectDir,
        refractiveIndex,
        rayIntensity,
        recursionDepth);
}
```

Chapter 11

Reorientable Solid Objects

11.1 The challenge of rotation

All objects drawn by this ray tracing code must be instances of a class derived, directly or indirectly, from `class SolidObject`. Any such derived class, as discussed earlier, must implement virtual functions for translating and rotating the object. In the case of `class Sphere`, the `Translate` function is simple: it just moves the center of the sphere, and that center point is used by any code that calculates surface intersections, surface normal vectors, inclusion of a point within the sphere, etc. Rotation of the sphere is trivial in the extreme: rotating a sphere about its center has no effect on its appearance, so `RotateX`, `RotateY`, and `RotateZ` don't have to do anything.

However, the mathematical derivation of other shapes can be much more complicated, even if we were to ignore the need for translating and rotating them. Once translation and rotation are thrown into the mix, the resulting equations can become overwhelming and headache-inducing. For example, as I was figuring out how to draw a torus (a donut shape), the added complexity of arbitrarily moving and spinning the torus soon led me to think, "there has to be a better way to do this." I found myself wishing I could derive the intersection formulas for a shape located at a fixed position and orientation in space, and have some intermediate layer of code abstract away the problem of translation and rotation. I soon realized that I could do exactly that by converting the position and direction vectors passed into methods like `AppendAllIntersections` to another system of coordinates, doing all the math in that other coordinate system, and finally converting any intersections found the inverse way, back to the original coordinate system. In simple terms, the effect would be to rotate and translate the point of view of the camera in the opposite way we want the object to be translated and rotated. For example, if you wanted to photograph your living room rotated 15° to the right, it would be much easier to rotate your camera 15° to the left than to actually rotate your living room, and the resulting image would be the same (except your stuff would not be falling over, of course).

11.2 The `SolidObject_Reorientable` class

This concept is implemented by the class `SolidObject_Reorientable`, which derives from `SolidObject`. `SolidObject_Reorientable` acts as a translation layer between two coordinate systems: the (x, y, z) coordinates we have been using all along, and a new set of coordinates (r, s, t) that are fixed from the point of view of a reorientable object. I will refer to (x, y, z) as *camera space* coordinates and (r, s, t) as *object space* coordinates. Each reorientable object has its own object space (r, s, t) coordinate system, independent of any other reorientable objects.

Class `SolidObject_Reorientable` introduces the following pure virtual methods that operate in object space, meaning they can be implemented with the assumption that the object resides at a fixed location and orientation with respect to an (r, s, t) coordinate system:

```
ObjectSpace_AppendAllIntersections
ObjectSpace_Contains
```

These two methods are just like `AppendAllIntersections` and `Contains` as defined in `class SolidObject`, only they are passed vector parameters that have already been translated

from camera coordinates to object coordinates. When an instance of a class derived from `SolidObject_Reorientable` is created, it starts out with (r, s, t) axes identical to the camera's (x, y, z) axes, as seen in Figure 11.1.

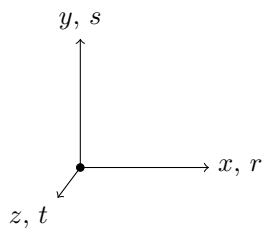

Figure 11.1: A reorientable object's (r, s, t) axes start out identical to the camera's (x, y, z) axes.

If the object is then translated, the (r, s, t) axes are shifted by the amounts specified by the translation vector, as show in Figure 11.2.

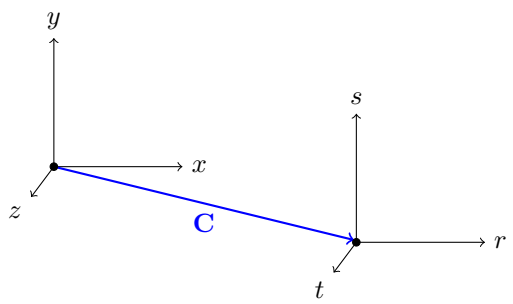

Figure 11.2: The (r, s, t) axes are shifted after the center of a reorientable object has been translated by the vector $\mathbf{C}$.

This is implemented by tracking the center of the object as usual, and shifting position vectors by subtracting the center position from the position of any (x, y, z) location to get the (unrotated) (r, s, t) coordinates.

If the object is rotated, the (r, s, t) axes are rotated with respect to the (x, y, z) axes. For example, `RotateZ(45.0)` applied to the example in Figure 11.2, where we had already translated the object, would cause the axes to look as shown in Figure 11.3.

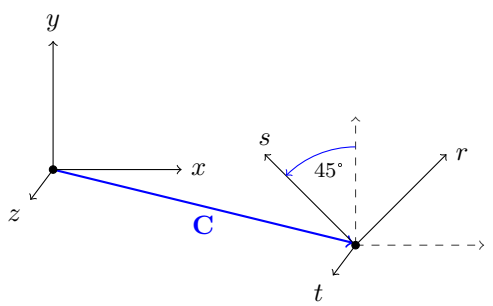

Figure 11.3: The (r, s, t) axes after having been rotated counterclockwise 45° parallel to the z axis.

The (r, s, t) axes have been rotated about the object's center, which is $(r, s, t) = (0, 0, 0)$ or $(x, y, z) = (C_x, C_y, C_z)$ by 45° counterclockwise parallel to the z axis, looking into the z axis, where $\mathbf{C} = (C_x, C_y, C_z)$ is the object's center as expressed in camera space.

11.3 Implementing a rotation framework

How is rotation implemented in `class SolidObject_Reorientable`? First, let's take a look at that class's declaration in `imager.h`. You will find that the `private` section of this class contains two triplets of `Vector`-typed member variables, one triplet being `rDir`, `sDir`, and `tDir`, the other being `xDir`, `yDir`, and `zDir`. All of these vectors are unit vectors. The first three, `rDir`, `sDir`, and `tDir`, hold the direction in (x, y, z) camera space that the (r, s, t) object space axes point. Conversely, `xDir`, `yDir`, and `zDir` hold the directions in (r, s, t) object space that the (x, y, z) camera space axes point. As mentioned above, a newly-created reorientable object starts out with (r, s, t) being identical to (x, y, z), so these member variable vectors are initialized as follows by the `SolidObject_Reorientable` constructor:

```
rDir=(1, 0, 0)    xDir=(1, 0, 0)
sDir=(0, 1, 0)    yDir=(0, 1, 0)
tDir=(0, 0, 1)    zDir=(0, 0, 1)
```

These are direction vectors, not position vectors, so they are not changed by the object being translated, but only when it is rotated. Translation affects only the center point position vector as reported by the protected member function `SolidObject::Center`. So the rotation problem breaks down into two questions:

1. How do we keep the six direction vectors up to date as various rotation operations are performed?

2. How do we use these six vectors to translate camera coordinates to object coordinates and object coordinates back to camera coordinates?

11.4 Complex numbers help rotate vectors

The first question is easier to resolve if we use a trick from complex number arithmetic: a complex number is a pair of numbers, the first called the *real* part, the second called the *imaginary* part. They are often written as $a + ib$, with a being the real part, b being the imaginary part, and i being the imaginary number $\sqrt{-1}$. If complex numbers are new to you, this idea probably seems bizarre and scary. But don't panic — for the purposes of this tutorial, there are only two things you need to know about complex numbers beyond basic algebra: first that $i^2 = -1$, and second that $a+ib$ can be thought of as just another way to write the two-dimensional vector (a, b). If we are rotating an object parallel to the z axis, we can pretend that all the object's points have their x and y components represented as complex numbers that get updated based on formulas I'm about to show you, and that their z components remain unchanged. Likewise, rotation parallel to the x axis involves x components remaining unchanged while y and z are changed as if they were complex numbers of the form $y + iz$. The components must be kept in the order consistent with the right-hand rule, as discussed in the section on vector cross products. (See Table 11.1.)

rotation operation	unchanged component	pretend complex number
`RotateX`	x	$y + iz = (y, z)$
`RotateY`	y	$z + ix = (z, x)$
`RotateZ`	z	$x + iy = (x, y)$

Table 11.1: Complex numbers used for rotation about x, y, z axes.

But how do complex numbers help us rotate two-dimensional vectors? I alluded to a trick, and here it is: when two complex numbers, say $a + ib$ and $c + id$ are multiplied, the product has the following relationship with the two complex numbers we started with:

1. The product's magnitude is equal to the product of the magnitudes of the two complex numbers.

2. The angle from the real axis of the product is equal to the sum of the angles of the two complex numbers.

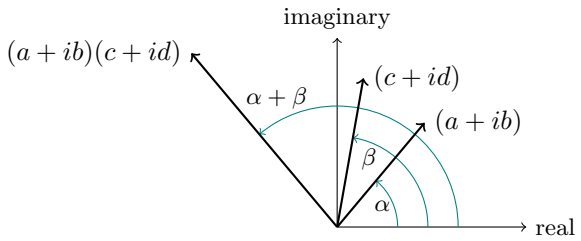

Figure 11.4: A geometric view of complex multiplication. The angle of $(a+ib)$ is α and the angle of $(c+id)$ is β. The complex product $(a+ib)(c+id)$ has angle $\alpha+\beta$, and its magnitude is the product of the magnitudes of $(a+ib)$ and $(c+id)$, or $\sqrt{a^2+b^2}\sqrt{c^2+d^2}$.

Figure 11.4 illustrates these two rules.

Multiplying complex numbers works using familiar rules of algebra:

$$(a+ib)(c+id) = ac + ibc + iad + i^2bd$$

As noted above, $i^2 = -1$, so we can write the product as

$$ac + ibd + iad - bd$$

Collecting the product terms into real and imaginary parts, we end up with

$$(a+ib)(c+id) = (ac - bd) + i(bc + ad)$$

Or, if you prefer, the product can be written using two-dimensional vector notation to represent the complex numbers:

$$(a, b)(c, d) = (ac - bd, bc + ad)$$

If we start with a two-dimensional vector (c, d) and we want to rotate it by an angle θ, all we need to do is to multiply it by the complex number (a, b) that represents that angle, so long as the magnitude of (a, b), or $\sqrt{a^2+b^2}$, is equal to 1. Here's how we calculate (a, b):

$$a = \cos(\theta)$$
$$b = \sin(\theta)$$

As we learned in Section 4.10, the point $(a, b) = (\cos(\theta), \sin(\theta))$ represents a point on a unit circle (a circle whose radius is one unit) at an angle θ measured counterclockwise from the positive horizontal axis. Because $|(a, b)| = \sqrt{a^2+b^2} = 1$, when we multiply (a, b) by (c, d), the product's magnitude will be the same as the magnitude of (c, d). And, as we wanted from the start, the product's angle will be rotated by an angle θ with respect to (c, d).

Depending on whether we are rotating parallel to the x, y, or z axis — that is, whether `RotateX`, `RotateY`, or `RotateZ` is being called, we will apply the complex multiplication formula to the other two components: if rotating parallel to x, the x component is not changed, but y and z are; if rotating parallel to y, the y component is not changed, but x and z are modified, etc. In any case, all three of the unit vectors `rDir`, `sDir`, `tDir` are updated using the same formula (whichever formula is appropriate for that type of rotation) as shown in Table 11.2, where (x, y, z) is the existing direction vector, and $(a, b) = (\cos(\theta), \sin(\theta))$.

old direction	operation	new direction
(x, y, z)	`RotateX`	$(x, ay - bz, by + az)$
(x, y, z)	`RotateY`	$(ax + bz, y, az - bx)$
(x, y, z)	`RotateZ`	$(ax - by, bx + ay, z)$

Table 11.2: Rotation formulas

11.5 Rotation matrices

We are left with the related issue of how to update the unit vectors `xDir`, `yDir`, and `zDir`, each expressing in (r, s, t) object coordinates how to convert vectors back into camera coordinates. This can be a confusing problem at first, because rotations are all parallel to the x, y, or z axis, not r, s, t. If we *were* to rotate about r, s, or t, we could use the same complex number tricks, but that is not the case — an example of the difficulty is that `RotateX` never changes to x component of `rDir`, `sDir`, or `tDir`, but it can change all of the components, r, s, and t, of `xDir`, `yDir`, and `zDir`.

But there is a surprisingly simple resolution of this confusing situation. Once we have calculated updated values for the unit vectors `rDir`, `sDir`, and `tDir`, we can arrange the three vectors, each having three components, in a 3-by-3 grid as shown in Table 11.3. Such a representation of

`rDir.x`	`rDir.y`	`rDir.z`
`sDir.x`	`sDir.y`	`sDir.z`
`tDir.x`	`tDir.y`	`tDir.z`

Table 11.3: Rotation matrix

three mutually perpendicular unit vectors has a name familiar to mathematicians and computer graphics experts: it is called a *rotation matrix*. Given these nine numbers, we want to calculate corresponding values for `xDir`, `yDir`, and `zDir`; these three unit vectors are known collectively as the *inverse rotation matrix*, since they will help us translate (r, s, t) object space vectors back into (x, y, z) camera space vectors. Remarkably, there is no need for any calculation effort at all; we merely need to rearrange the nine numbers from the original rotation matrix to obtain the inverse rotation matrix. The rows of the original matrix become the columns of the inverse matrix, and vice versa, as shown in Table 11.4. This rearrangement procedure is called *transpo-*

`rDir.x`	`sDir.x`	`tDir.x`
`rDir.y`	`sDir.y`	`tDir.y`
`rDir.z`	`sDir.z`	`tDir.z`

Table 11.4: Inverse rotation matrix

sition — we say that the inverse rotation matrix is the transpose of the original rotation matrix. So after we update the values of `rDir`, `sDir`, and `tDir` in `RotateX`, `RotateY`, or `RotateZ`, we just need to transpose the values into `xDir`, `yDir`, and `zDir`:

```
xDir = Vector(rDir.x, sDir.x, tDir.x);
yDir = Vector(rDir.y, sDir.y, tDir.y);
zDir = Vector(rDir.z, sDir.z, tDir.z);
```

This block of code, needed by all three rotation methods, is encapsulated into the member function `SolidObject_Reorientable::UpdateInverseRotation`, which is located in `imager.h`. The code for `RotateX`, `RotateY`, and `RotateZ` is all located in `reorient.cpp`.

A point of clarification is helpful here. Although the vectors `xDir`, `yDir`, and `zDir` are object space vectors, and therefore contain (r, s, t) components mathematically, they are implemented using the same C++ class `Vector` as is used for camera space vectors. This C++ class has members called `x`, `y`, and `z`, but we must understand that when `class Vector` is used for object space vectors, the members are to be interpreted as (r, s, t) components. This implicit ambiguity eliminates otherwise needless duplication of code, avoiding a redundant version of `class Vector` (and its associated inline functions and operators) having members named `r`, `s`, and `t`.

11.6 Translating between camera coordinates and object coordinates

At this point, we have completely resolved the problem of updating the six unit direction vectors as needed for calls to `RotateX`, `RotateY`, and `RotateZ`. Now we return to the second question: how do we use these vectors to translate back and forth between camera space and object space?

The answer is: we use vector dot products. But we must take care to distinguish between vectors that represent directions and those that represent positions. Direction vectors are simpler, so we consider them first. Let $\mathbf{D} = (x, y, z)$ be a direction vector in camera space. We wish to find the object space direction vector $\mathbf{E} = (r, s, t)$ that points in the same direction. We already have computed the unit vectors `rDir`, `sDir`, and `tDir`, which specify (x, y, z) values showing which way the r axis, s axis, and t axis point. To calculate r, the r-component of the object space direction vector $\mathbf{E}$, we take advantage of the fact that the dot product of a direction vector with a unit vector tells us how much the direction vector extends in the direction of the unit vector. We take $\mathbf{D}\cdot$`rDir` as the value of r, thinking of that dot product as representing the "shadow" of $\mathbf{D}$ onto the r axis.

The same analysis applies to s and t, yielding the equations

$$r = \mathbf{D} \cdot \mathrm{rDir}$$
$$s = \mathbf{D} \cdot \mathrm{sDir}$$
$$t = \mathbf{D} \cdot \mathrm{tDir}$$

If you look inside `class SolidObject_Reorientable` in the header file `imager.h`, you will find the following function that performs these calculations:

```
Vector ObjectDirFromCameraDir(const Vector& cameraDir) const
{
    return Vector(
        DotProduct(cameraDir, rDir),
        DotProduct(cameraDir, sDir),
        DotProduct(cameraDir, tDir)
    );
}
```

To compute the inverse, that is, a camera direction vector from an object direction vector, we use the function `CameraDirFromObjectDir`, which is almost identical to `ObjectDirFromCameraDir`, only it computes dot products with `xDir`, `yDir`, `zDir` instead of `rDir`, `sDir`, `tDir`.

Dealing with position vectors (that is, points) is only slightly more complicated than direction vectors. They require us to adjust for translations the object has experienced, based on the center position reported by the member function `SolidObject::Center`. Here are the corresponding functions for converting points from camera space to object space and back:

```
Vector ObjectPointFromCameraPoint(const Vector& cameraPoint) const
{
    return ObjectDirFromCameraDir(cameraPoint - Center());
}

Vector CameraPointFromObjectPoint(const Vector& objectPoint) const
{
    return Center() + CameraDirFromObjectDir(objectPoint);
}
```

Now that all the mathematical details of converting back and forth between camera space and vector space have been taken care of, it is fairly straightforward to implement `SolidObject_Reorientable` methods like `AppendAllIntersections`, shown here as it appears in `reorient.cpp`:

```
void SolidObject_Reorientable::AppendAllIntersections(
    const Vector& vantage,
    const Vector& direction,
    IntersectionList& intersectionList) const
{
    const Vector objectVantage = ObjectPointFromCameraPoint(vantage);
    const Vector objectRay     = ObjectDirFromCameraDir(direction);

    const size_t sizeBeforeAppend = intersectionList.size();

    ObjectSpace_AppendAllIntersections(
```

```
        objectVantage,
        objectRay,
        intersectionList
    );

    // Iterate only through the items we just appended,
    // skipping over anything that was already in the list
    // before this function was called.
    for (size_t index = sizeBeforeAppend;
         index < intersectionList.size();
         ++index)
    {
        Intersection& intersection = intersectionList[index];

        // Need to transform intersection back into camera space.
        intersection.point =
            CameraPointFromObjectPoint(intersection.point);

        intersection.surfaceNormal =
            CameraDirFromObjectDir(intersection.surfaceNormal);
    }
}
```

Note that this function translates parameters expressed in camera coordinates to object coordinates, calls `ObjectSpace_AppendAllIntersections`, and converts resulting object space vectors (intersection points and surface normal vectors) back into camera space, as the caller expects.

11.7 Simple example: the `Cuboid` class

To illustrate how to use `SolidObject_Reorientable` as a base class, let's start with a very simple mathematical shape, the cuboid. A cuboid is just a rectangular box: it has six rectangular faces, three pairs of which are parallel and at right angles to the remaining pairs. The cuboid's length, width, and height may have any desired positive values.

We use a to refer to half of the cuboid's width, b for half of its length, and c for half its height, so its dimensions are $2a$ by $2b$ by $2c$. This approach simplifies the math. For example, the left face is at $r = -a$, the right face is at $r = +a$, etc. Note that using object space coordinates r, s, and t allows us to think of the cuboid as locked in place, with the (r, s, t) origin always at the center of the cuboid and the three axes always orthogonal with the cuboid's faces.

The class `Cuboid` is declared in the header file `imager.h` as follows:

```
// A "box" with rectangular faces, all of which are mutually perpendicular.
class Cuboid: public SolidObject_Reorientable
{
public:
    Cuboid(double _a, double _b, double _c, Color _color)
        : SolidObject_Reorientable()
        , color(_color)
        , a(_a)
        , b(_b)
        , c(_c)
    {
    }

    Cuboid(double _a, double _b, double _c,
           Color _color, const Vector& _center)
        : SolidObject_Reorientable(_center)
        , color(_color)
        , a(_a)
        , b(_b)
        , c(_c)
    {
    }

protected:
    virtual size_t ObjectSpace_AppendAllIntersections(
        const Vector& vantage,
        const Vector& direction,
```

```
        IntersectionList& intersectionList) const;

    virtual bool ObjectSpace_Contains(const Vector& point) const
    {
        return
            (fabs(point.x) <= a + EPSILON) &&
            (fabs(point.y) <= b + EPSILON) &&
            (fabs(point.z) <= c + EPSILON);
    }

private:
    const Color     color;
    const double    a;      // half of the width
    const double    b;      // half of the length
    const double    c;      // half of the height
};
```

The `Cuboid` class is a good example of the minimal steps needed to create a reorientable solid class:

1. The class derives publicly from `SolidObject_Reorientable`.
2. It provides constructor(s) to initialize dimensions, color, etc.
3. It provides implementations of the following methods:
   - `ObjectSpace_AppendAllIntersections`
   - `ObjectSpace_Contains`
4. It declares whatever private member variables are needed to hold dimensions, color, etc.

The `ObjectSpace_Contains` function is very simple thanks to fixed object space coordinates: a point is inside the cuboid if $-a \leq r \leq +a$, $-b \leq s \leq +b$, and $-c \leq t \leq +c$. Because the function's `point` parameter is of type `Vector`, the C++ code requires us to type `point.x`, `point.y`, and `point.z`, but we understand these to refer to the point's r, s, and t components. Also, to provide tolerance for floating point rounding errors, we widen the ranges of inclusion by the small amount `EPSILON`. This method will be used in `ObjectSpace_AppendAllIntersections`, and using `EPSILON` like this helps ensure that we correctly include intersection points in the result. (The extra tolerance will also be necessary to use `Cuboid` as part of a set operation, a topic that is covered in a later chapter.) Using the standard library's absolute value function `fabs`, we therefore express $-a \leq r \leq +a$ as

```
fabs(point.x) <= a + EPSILON
```

and so on.

Coding the method `Cuboid::ObjectSpace_AppendAllIntersections` requires the same kind of mathematical approach we used in `Sphere::AppendAllIntersections`:

1. We write a parametric ray equation, only this time in terms of (r, s, t) coordinates instead of (x, y, z) coordinates. So $\mathbf{P} = (r, s, t) = \mathbf{D} + u\mathbf{E}$, where $\mathbf{D}$ is a vantage point and $\mathbf{E}$ is a direction vector, both expressed in object space coordinates.
2. We write equations for the solid's surfaces. In the case of the cuboid, the bad news is that there are six surfaces instead of the sphere's single surface. But the good news is that each of the equations is simple and linear, not quadratic as was the case with the sphere. (See Table 11.5.)

   After solving the appropriate surface equation to find a surface point $\mathbf{P} = (r, s, t)$, we need to determine whether the values r, s, t are actually bounded by the finite size of the cuboid. To do this, we call `ObjectSpace_Contains`. If it returns `true`, we know the point is actually on the cuboid's surface; otherwise, it isn't.
3. We find the intersection (if any) of the parametric ray equation and the cuboid's faces. Let's work through the solution of the left face equation $r = -a$ and how the ray $\mathbf{P} = \mathbf{D} + u\mathbf{E}$ might intersect with it, as an example. The other five faces will work the same way.

$$\begin{aligned}\mathbf{P} &= \mathbf{D} + u\mathbf{E} \\ &= (D_r, D_s, D_t) + u(E_r, E_s, E_t)\end{aligned}$$

face	equation
left	$r = -a$
right	$r = +a$
front	$s = -b$
back	$s = +b$
bottom	$t = -c$
top	$t = +c$

Table 11.5: Cuboid surface equations

For the point **P** to be on the left face, its r component must be equal to $-a$, so:

$$D_r + uE_r = -a$$

Solving for the parameter u, we find

$$u = \frac{-a - D_r}{E_r}, \quad E_r \neq 0, \quad u > 0.$$

Substituting u back into the parametric ray equation, we find the values of the s and t components of **P**, completely determining the point **P**:

$$\mathbf{P} = (-a,\, D_s + uE_s,\, D_t + uE_t)$$

If we find that $s = D_s + uE_s$ is in the range $-b...+b$ and that $t = D_t + uE_t$ is in the range $-c...+c$ (that is, that `ObjectSpace_Contains(P)` returns `true`), we know that the intersection point is genuinely on the cuboid's left face. As always, we have to check that $u >$ `EPSILON`, ensuring that the intersection point is significantly in the intended direction **E** from the vantage point **D**, not at **D** or in the opposite direction $-\mathbf{E}$.

4. When we find a point on the cuboid's left face (or any other face) we must determine the surface normal unit vector in (r, s, t) form. No calculation is necessary because the answer is the same for any point on the cuboid's left face: it is a unit vector pointing in the $-r$ direction, or $(-1, 0, 0)$.

The code for the remaining five faces is very similar, as you can see by looking at `cuboid.cpp`.

11.8 Another reorientable solid: the `Cylinder` class

Let's take a look at another reorientable solid: the `Cylinder` class. A cylinder is like an idealized soup can, consisting of three surfaces: a curved lateral tube and two circular discs, one on the top and another on the bottom. This time I will cover the mathematical solution for finding intersections between a ray and a cylinder and for the surface normal vector at an intersection, but I will omit discussion of the coding details, as they are very similar to what we already covered in `class Cuboid`.

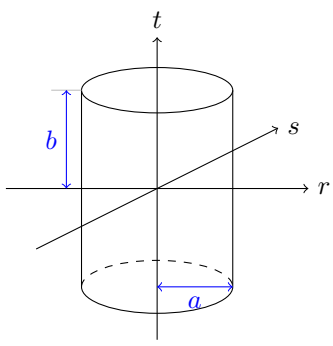

Figure 11.5: Cylinder with radius a and half-height b

surface	equation	constraint
top disc	$t = +b$	$r^2 + s^2 \leq a^2$
bottom disc	$t = -b$	$r^2 + s^2 \leq a^2$
tube	$r^2 + s^2 = a^2$	$-b \leq t \leq +b$

Table 11.6: Cylinder surface equations and constraints

The cylinder is aligned along the t axis, with its top disc located at $t = +b$ and its bottom disc at $t = -b$. The discs and the lateral tube have a common radius a. These three surfaces have the equations and constraints as shown in Table 11.6.

The tube equation derives from taking any cross-section of the cylinder parallel to the rs plane and between the top and bottom discs. You would be cutting through a perfect circle: the set of points in the cross-sectional plane that measure a units from the t axis. The value of t where you cut is irrelevant — you simply have the formula for a circle of radius a on the rs plane, or $r^2 + s^2 = a^2$.

As usual, we find the simultaneous solution of the parametric ray equation with each of the surface equations, yielding potential intersection points for the corresponding surfaces. Each potential intersection is valid only if it satisfies the constraint for that surface.

The equations for the discs are simpler than the tube equation, so we start with them. We can write both equations with a single expression:

$$t = \pm b$$

Then we combine the disc surface equations with the parametric ray equation:

$$\begin{aligned} \mathbf{P} &= \mathbf{D} + u\mathbf{E} \\ (r, s, t) &= (D_r, D_s, D_t) + u(E_r, E_s, E_t) \\ (r, s, t) &= (D_r + uE_r, D_s + uE_s, D_t + uE_t) \end{aligned}$$

In order for a point $\mathbf{P}$ on the ray to be located on one of the cylinder's two discs, the t component must be either $-b$ or $+b$:

$$D_t + uE_t = \pm b$$

Solving for u, we find

$$u = \frac{\pm b - D_t}{E_t}, \quad E_t \neq 0$$

Assuming $u > 0$, the potential intersection points for the discs are

$$\left(D_r + \left(\frac{\pm b - D_t}{E_t} \right) E_r \ , \ D_s + \left(\frac{\pm b - D_t}{E_t} \right) E_s \ , \ \pm b \right)$$

Letting $r = D_r + \left(\frac{\pm b - D_t}{E_t}\right) E_r$ and $s = D_s + \left(\frac{\pm b - D_t}{E_t}\right) E_s$, we check the constraint $r^2 + s^2 \leq a^2$ to see if each value of $\mathbf{P}$ is within the bounds of the disc. If so, the surface normal vector is $(0, 0, 1)$ for any point on the top disc, or $(0, 0, -1)$ for any point on the bottom disc — a unit vector perpendicular to the disc and outward from the cylinder in either case.

We turn our attention now to the problem of finding intersections of a ray with the lateral tube. Just as in the case of a sphere, a ray may intersect a tube in two places, so we should expect to end up having a quadratic equation. We find that the simultaneous solution of the parametric ray equation and the surface equation for the tube gives us exactly that:

$$\begin{cases} (r, s, t) = (D_r + uE_r, D_s + uE_s, D_t + uE_t) \\ r^2 + s^2 = a^2 \end{cases}$$

We substitute the values $r = D_r + uE_r$ and $s = D_s + uE_s$ from the ray equation into the tube equation:

$$(D_r + uE_r)^2 + (D_s + uE_s)^2 = a^2$$

The only unknown quantity in this equation is u, and we wish to solve for it in terms of the other quantities. The algebra proceeds as follows:

$$(D_r^2 + 2D_rE_ru + E_r^2u^2) + (D_s^2 + 2D_sE_su + E_s^2u^2) = a^2$$

$$(E_r^2 + E_s^2)u^2 + 2(D_r E_r + D_s E_s)u + (D_r^2 + D_s^2 - a^2) = 0$$

Letting

$$\begin{cases} A = E_r^2 + E_s^2 \\ B = 2(D_r E_r + D_s E_s) \\ C = D_r^2 + D_s^2 - a^2 \end{cases}$$

we have a quadratic equation in standard form:

$$Au^2 + Bu + C = 0$$

As in the case of the sphere, we try to find real and positive values of u via

$$u = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

If $B^2 - 4AC < 0$, we give up immediately. The imaginary result of taking the square root of a negative number is our signal that the ray does not intersect with an infinitely long tube of radius r aligned along the t axis, let alone the finite cylinder. But if $B^2 - 4AC \geq 0$, we use the formula above to find two real values for u. As before, there are two extra hurdles to confirm that the point $\mathbf{P}$ is on the tube: u must be greater than 0 (or `EPSILON` in the C++ code, to avoid rounding error problems) and $t = D_t + uE_t$ must be in the range $-b \leq t \leq +b$ (or $|t| \leq b+$ `EPSILON`, again to avoid roundoff error problems).

11.9 Surface normal vector for a point on the tube surface

When we do confirm that an intersection point lies on the tube's surface, we need to calculate the surface normal unit vector at that point. To be perpendicular to the tube's surface, the normal vector must point directly away from the center of the tube, and be parallel with the top and bottom discs.

The normal vector $\hat{\mathbf{n}}$ must lie in a plane that includes the intersection point $\mathbf{P}$ and that is parallel to the rs plane. As such, $\hat{\mathbf{n}}$ is perpendicular to the t axis, so its t component is zero. If we let $\mathbf{Q}$ be the point along the t axis that is closest to the intersection point $\mathbf{P} = (P_x, P_y, P_z)$, we can see that $\mathbf{Q} = (0, 0, P_z)$. The vector difference $\mathbf{P} - \mathbf{Q} = (P_x, P_y, 0)$ points in the same direction as $\hat{\mathbf{n}}$, but has magnitude $|\mathbf{P} - \mathbf{Q}| = a$, where a is the radius of the cylinder. To convert to a unit vector, we must divide the vector difference by its magnitude:

$$\hat{\mathbf{n}} = \frac{\mathbf{P} - \mathbf{Q}}{a} = \left(\frac{P_x}{a}, \frac{P_y}{a}, 0\right)$$

We now have a complete mathematical solution for rendering a ray-traced image of a cylinder. The C++ implementation details are all available in the declaration for `class Cylinder` in `imager.h` and the function bodies in `cylinder.cpp`.

Chapter 12

The `TriangleMesh` class

12.1 Making things out of triangles

The C++ class `TriangleMesh` derives from `SolidObject`. It represents a solid object whose surfaces are all triangles. A simple example of such a solid is the four-sided regular polyhedron called the *tetrahedron* (see Figure 12.1).

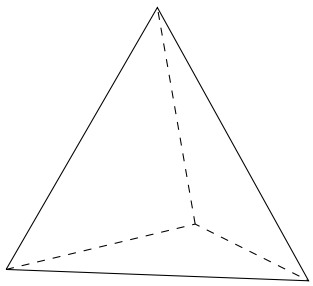

Figure 12.1: A tetrahedron has four equilateral triangles as faces.

Creating a visible solid like the tetrahedron proceeds with the calling code following steps like these:

1. Dynamically allocate an instance of `TriangleMesh` using `operator new`.

2. Call the instance's `AddPoint` method to define all the vertex points on the solid (these are the corner points of the triangular faces). Note that a single vertex can be shared by multiple triangular faces. For example, each of the vertices in a tetrahedron is shared by three of the four triangles. Every call to `AddPoint` must include an integer index as its first parameter. The first call must pass the value 0, the second muss pass 1, etc. This parameter is called `pointIndex`, and serves as a sanity check for the programmer, primarily because subsequent code will need to refer to these point indices. If the wrong value is passed into `AddPoint`, it will throw an exception.

3. Call `AddTriangle` for each triangular face to be added to the solid. Pass three distinct point indices to specify the vertex points that define the bounary of the new triangle.

4. Add the `TriangleMesh` instance to the `Scene` so that it is included in the rendered image.

12.2 Intersection of a ray and a triangle's plane

I will explain here how the math works for finding an intersection of a ray with a single triangle. `TriangleMesh` applies the same math to every triangle it contains when its `AppendAllIntersections` method is called. Each valid intersection gets appeneded to the output parameter `intersectionList`. Given a triangle whose vertices are **A**, **B**, and **C**, we wish to know whether a ray starting at vantage point **D** and pointing in direction **E** passes through the

plane of triangle **ABC**, and if so, where the intersection point **P** is. (See Figure 12.2.) We will determine in the next section whether **P** is actually inside the triangle or not, but first things first.

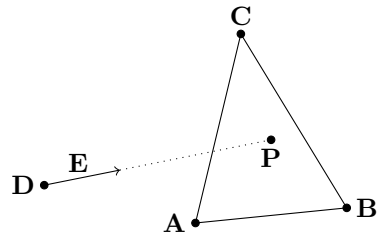

Figure 12.2: An intersection of a ray and a triangle.

As usual, we start with the parametric ray equation

$$\mathbf{P} = \mathbf{D} + u\mathbf{E}$$
$$(P_x, P_y, P_z) = (D_x + uE_x, D_y + uE_y, D_z + uE_z)$$

Now we have to write parametric equations for every point **P** on the plane of the triangle **ABC**. Because any plane is a two-dimensional surface, we will need a parametric equation with two unknowns as parameters. Let's call the new parameters v and w. How do we construct a parametric equation that allows us to sweep through every point on the plane passing through the points **A**, **B**, and **C**, just by varying the values of the scalar parameters v and w? There are many possibilities, but here is the approach I took. I pretend that an axis passing through **A** and **B** is the "v axis" and an axis passing through **A** and **C** is the "w axis". Then each point on the plane has a location with a v coordinate and a w coordinate. At first it may seem wrong to have a coordinate system where the two axes, v and w, are not required to be perpendicular. But if we are careful to make no assumptions that require perpendicular axes, things will work out just fine. Figure 12.3 shows how this coordinate system can specify an arbitrary point on the plane that passes through three points **A**, **B**, and **C**.

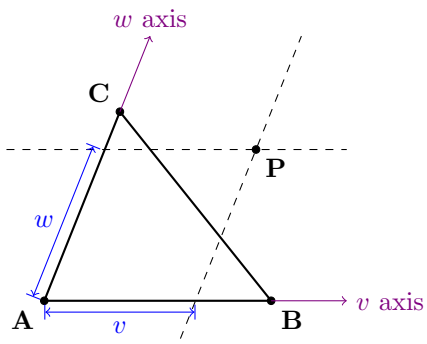

Figure 12.3: Any point **P** on a plane **ABC** can be specified using parameters v and w.

We extend a line starting at **A** and proceeding through **B**. This line extends infinitely along that direction, and becomes our v axis. Likewise, the w axis starts at **A** and extends through **C** and off to infinity. For any point **P** on the same plane as **ABC**, we can draw lines through **P** that are parallel to the v and w axes, as indicated by the dashed lines in Figure 12.3. The line passing through **P** parallel to the v axis will pass through the w axis somewhere, and the one parallel to the w axis will pass through the v axis at another location. We can think of these locations along the v and w axes as "shadows" of the point **P** falling onto them. I will define the value v as the v component of **P** (**P**'s shadow on the v axis) and w as the w component of **P** (**P**'s shadow on the w axis). Furthermore, v can be thought of as the fraction of the distance along the line segment **AB** that the shadow lies, such that **A** is at $v = 0$ and **B** is at $v = 1$. For example, if the shadow of **P** lies one fourth of the distance from **A** to **B**, we say that $v = 0.25$. Negative numbers work fine also: if the shadow of **P** on the v axis is the same distance from **A** as **B** is, but is in the opposite direction as **B**, we say that $v = -1$. If **P**'s shadow on the v axis

is in the same direction as **B** from **A**, but is twice as far away, we say that $v = 2$. So v may have any value from $-\infty$ to $+\infty$, depending on where **P** is on the infinite plane passing through the triangle **ABC**. Analogously, w is defined as the fraction along the line segment **AC** that the shadow of **P** measures along the w axis.

In vector notation, we can think of v and w as scalars that we multiply with the vectors from **A** to **B** and from **A** to **C**, respectively, to get displacement vectors from **A** along those two axes. Any point **P** is the sum of these two displacement vectors along the two axes, plus the starting point **A**:

$$\mathbf{P} = \mathbf{A} + v(\mathbf{B} - \mathbf{A}) + w(\mathbf{C} - \mathbf{A})$$

Let's try some examples to test whether this equation makes sense. If we choose **P** to be at the same location as **B**, we expect v, the fraction from **A** to **B** along the v axis, to be 1. Likewise, we expect w to be 0, since we don't travel at all along the w axis. Plugging $v = 1$, $w = 0$ into the equation, we find:

$$\begin{aligned}
\mathbf{P} &= \mathbf{A} + 1(\mathbf{B} - \mathbf{A}) + 0(\mathbf{C} - \mathbf{A}) \\
\mathbf{P} &= \mathbf{A} + (\mathbf{B} - \mathbf{A}) \\
\mathbf{P} &= \mathbf{B}
\end{aligned}$$

So that case works. As another example, to select $\mathbf{P} = \mathbf{C}$, we can test with $v = 0$, $w = 1$:

$$\begin{aligned}
\mathbf{P} &= \mathbf{A} + 0(\mathbf{B} - \mathbf{A}) + 1(\mathbf{C} - \mathbf{A}) \\
\mathbf{P} &= \mathbf{A} + (\mathbf{C} - \mathbf{A}) \\
\mathbf{P} &= \mathbf{C}
\end{aligned}$$

That works too. And if both v and w are 0, we have not moved from **A** at all:

$$\begin{aligned}
\mathbf{P} &= \mathbf{A} + 0(\mathbf{B} - \mathbf{A}) + 0(\mathbf{C} - \mathbf{A}) \\
\mathbf{P} &= \mathbf{A}
\end{aligned}$$

The parametric ray equation

$$\mathbf{P} = \mathbf{D} + u\mathbf{E}$$

and the parametric plane equation

$$\mathbf{P} = \mathbf{A} + v(\mathbf{B} - \mathbf{A}) + w(\mathbf{C} - \mathbf{A})$$

describe the set of all points on a ray and the set of all points on a plane, respectively. If the ray intersects with the plane, both equations must refer to the same point **P** where the ray meets the plane, so we can set the right-hand sides of the equations equal to each other:

$$\mathbf{D} + u\mathbf{E} = \mathbf{A} + v(\mathbf{B} - \mathbf{A}) + w(\mathbf{C} - \mathbf{A})$$

At first, this seems insoluble, because there are three unknowns (u, v, w) but only 1 equation. But this equation is a vector equation in three spatial dimensions, so we can rewrite it:

$$\begin{aligned}
(D_x + uE_x, D_y + uE_y, D_z + uE_z) = {} & (A_x, B_x, C_x) \\
& + v(B_x - A_x, B_y - A_y, B_z - A_z) \\
& + w(C_x - A_x, C_y - A_y, C_z - A_z)
\end{aligned}$$

This becomes three linear scalar equations in three unknowns:

$$\begin{cases}
D_x + uE_x = A_x + v(B_x - A_x) + w(C_x - A_x) \\
D_y + uE_y = A_y + v(B_y - A_y) + w(C_y - A_y) \\
D_z + uE_z = A_z + v(B_z - A_z) + w(C_z - A_z)
\end{cases}$$

We can rearrange the terms to arrive at the usual form for systems of linear equations, with each unknown term having a known coefficient on the left and the unknown variable on the right, and a constant term, all adding up to zero:

$$\begin{cases}
0 = E_x u + (A_x - B_x)v + (A_x - C_x)w + (D_x - A_x) \\
0 = E_y u + (A_y - B_y)v + (A_y - C_y)w + (D_y - A_y) \\
0 = E_z u + (A_z - B_z)v + (A_z - C_z)w + (D_z - A_z)
\end{cases}$$

I will not go into the details of solving this system of equations, because this is a topic in algebra that is covered well by many other authors. If you look for class `TriangleMesh` in `imager.h`, you will see that `AttemptPlaneIntersection` is one of its member functions. It tries to solve the above system of equations using the utility method `SolveLinearEquations` that is declared in `algebra.h` and implemented in `algebra.cpp`.

I say it tries to solve the system, because there are special cases where the solver cannot find a solution because it must avoid dividing by zero. This can happen, for example, if the ray is parallel to either $\mathbf{B} - \mathbf{A}$ or $\mathbf{C} - \mathbf{A}$. In `triangle.cpp`, you will see that `TriangleMesh::AppendAllIntersections` will try the points of a triangle in up to three different orders, $(\mathbf{A}, \mathbf{B}, \mathbf{C})$, $(\mathbf{B}, \mathbf{C}, \mathbf{A})$, and $(\mathbf{C}, \mathbf{A}, \mathbf{B})$, because sometimes one or two of the orders can fail but another succeeds.

12.3 Determining whether a point is inside the triangle

If we find a triplet of values u, v, w that solve the system of equations, it just means that the point $\mathbf{P} = \mathbf{D} + u\mathbf{E}$ on the ray is also on the plane passing through the triangle $\mathbf{ABC}$. As always, we have to check for $u > 0$ to determine whether $\mathbf{P}$ lies in the intended direction $\mathbf{E}$ from the vantage point $\mathbf{D}$. But we also must validate that v and w refer to a point inside the triangle $\mathbf{ABC}$ itself, not just some point on the infinite plane passing through that triangle.

Based on our earlier discussion of what the parameters v and w mean, we know that both must be in the range 0 to 1, but that is not sufficient for ensuring that the point is inside the triangle. Let's take a look at the region of the plane covered by constraining $0 \leq v \leq 1$ and $0 \leq w \leq 1$, shown in Figure 12.4.

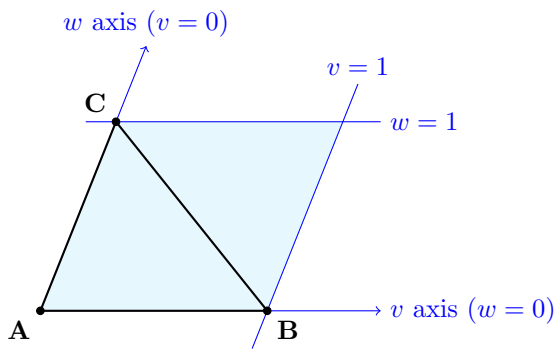

Figure 12.4: The shaded region is the set of all points where $0 \leq v \leq 1$ and $0 \leq w \leq 1$.

The region of the plane between the horizontal lines $w = 0$ and $w = 1$ is an infinitely wide horizontal strip, each point of which has a value of w between 0 and 1. Likewise, the slanted strip between the slanted lines $v = 0$ and $v = 1$ is the set of all points where v is between 0 and 1. These two regions each have an infinite area, but they overlap in a finite extent: a parallelogram as shown by the shaded region in Figure 12.4. This parallelogram encompasses all points where $0 \leq v \leq 1$ and $0 \leq w \leq 1$. But it includes twice as much area as the intended triangle $\mathbf{ABC}$. What we really want is a collection of three constraints: to be inside the triangle, a point must be above the line passing through $\mathbf{A}$ and $\mathbf{B}$ (i.e., above the v axis), to the right of the line passing through $\mathbf{A}$ and $\mathbf{C}$ (to the right of the w axis) and to the left of the line passing through $\mathbf{B}$ and $\mathbf{C}$.

How do we express this third constraint in terms of v and w? As we have seen before, if we let $v = 0$ and $w = 1$, then $\mathbf{P}$ is at the same location as $\mathbf{C}$. If we let $v = 1$ and $w = 0$, $\mathbf{P}$ is at $\mathbf{B}$. But what happens to v and w as $\mathbf{P}$ moves from $\mathbf{B}$ along the line toward $\mathbf{C}$? We can see that v gradually increases from 0 to 1 while w gradually decreases from 1 down to 0. Noting that at the extreme points $\mathbf{B}$ and $\mathbf{C}$, $1 + 0 = 0 + 1 = 1$, and reasoning that both v and w change linearly along the line segment $\mathbf{BC}$, we can deduce that $v + w = 1$ all along the line segment. So if $v + w = 1$, we know that $\mathbf{P}$ is on the line $\mathbf{BC}$. If $v + w < 1$, we know that $\mathbf{P}$ is on the side of $\mathbf{BC}$ closer to $\mathbf{A}$. If $v + w > 1$, then $\mathbf{P}$ is on the side of $\mathbf{BC}$ away from $\mathbf{A}$. So for $\mathbf{P}$ to be on the boundary of the triangle or inside the triangle, we must have $v + w \leq 1$. So in summary, we

have the following four constraints:

$$\begin{cases} u > 0 \\ v \geq 0 \\ w \geq 0 \\ v + w \leq 1 \end{cases}$$

When we find u, v, w that satisfy all four constraints, we have confirmed that the point $\mathbf{P}$ is inside the triangle and is along the ray in the intended direction. At this time we no longer need v and w; we can just use u to find the location of the intersection point $\mathbf{P} = \mathbf{D} + u\mathbf{E}$.

12.4 The surface normal vector of a triangle

To find the surface normal vector for a point on a triangle, all we need to know is the triangle itself, not where the point is. This is because a triangle is a completely flat surface, so the same direction is perpendicular everywhere on the triangle (or everywhere on the infinite plane coinciding with the triangle, for that matter). What we need is a formula that computes the surface normal vector from the three vertices of the triangle, $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$. If we could find two vectors that were both in the plane of the triangle, but pointing in different directions, we could calculate their vector cross product. The resulting vector would be perpendicular to both of the vectors we started with, and therefore perpendicular to the entire plane of the triangle. Dividing the cross product by its own magnitude gives us a unit vector, and it points in a direction perpendicular to the triangular surface. And we can easily find two vectors in the plane of the triangle using vector subtraction: $\mathbf{B} - \mathbf{A}$ is the vector pointing from $\mathbf{A}$ to $\mathbf{B}$, and $\mathbf{C} - \mathbf{B}$ is the vector pointing from $\mathbf{B}$ to $\mathbf{C}$.

$$\begin{aligned} &\text{Let} \quad \mathbf{N} = (\mathbf{B} - \mathbf{A}) \times (\mathbf{C} - \mathbf{B}). \\ &\text{Then} \quad \hat{\mathbf{n}} = \frac{\mathbf{N}}{|\mathbf{N}|}. \end{aligned}$$

12.5 An ambiguity: which perpendicular is correct?

One problem remains: depending on which order we choose $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ for the vertices of a triangle, we will end up with a surface normal unit vector $\hat{\mathbf{n}}$ that points in one of two opposite directions. The correct direction is the one that points outward from the body of the solid. But this begs the question: all we know is a triangle's vertices; it is ambiguous to consider one perpendicular direction to be its "inside" and the other its "outside."

In fact, thinking about the solid's inside or outside makes sense only in the context of an entire collection of triangles enclosing a volume of space. The distinction between the solid's inside and outside breaks down if the triangles do not enclose one or more volumes of space, all in an airtight manner. If there are any leaks or holes due to missing or misplaced triangular faces, light rays can leak into the interior of the solid and illuminate faces from the "wrong" side. The dot product formula in `Scene::CalculateLighting` will have a negative value, and therefore the surface will be considered to be in shadow with respect to the originating light source. Making the ray tracer C++ code enforce that a `TriangleMesh` instance always fully enclose its interior is possible, but it is a very complex problem. I decided the negative consequence of a leak (an image that doesn't look right) was not worth the extra complexity in educational code like this. Instead, it is the responsibility of the programmer to design a fully-enclosed `TriangleMesh` by providing it with a correct series of `AddPoint` and `AddTriangle` calls.

Even so, we are left with the problem of choosing which of the two opposite orientations of a surface normal vector is correct for pointing outward from the solid's interior. There is an algorithm that could figure this out in theory: tracing rays in both directions, counting how many surfaces are hit; an even number (0, 2, ...) indicates that direction points outward from the solid, and an odd number of intersections means that direction points inward. But this is a lot of computational work, and there are pitfalls that make it difficult to write code that works in all cases. To make the code much faster and simpler, I put the burden of selecting the outward normal direction for each triangle on the code that calls `TriangleMesh::AddTriangle`. Such code may order the three point indices in six possible ways. For example, if the indices are 0, 1, and 2, the following orderings are possible to add the triangle:

- 0, 1, 2
- 0, 2, 1
- 1, 0, 2
- 2, 0, 1
- 1, 2, 0
- 2, 1, 0

The rule is: the caller of `AddTriangle` chooses the ordering of points such that they appear in a counterclockwise arrangement as seen from the intended "outside" of the entire solid. So three of the six orderings listed above will result in the surface normal pointing in one direction, and the other three will cause it to point in the opposite direction. Don't worry too much about making a mistake here; if you do, the incorrectly-oriented triangles will be very obviously solid black instead of their intended colors. When this happens, swap any two of the indices in the `AddTriangle` calls for those triangles and they will start rendering correctly.

12.6 Example: using `TriangleMesh` to draw an icosahedron

An example of using the `TriangleMesh` class is provided in `icosahedron.cpp`: the class `Icosahedron` derives from class `TriangleMesh`, and its constructor shows how to call `AddPoint` and `AddTriangle` to build a 20-sided regular polyhedron. When you run the `raytrace` unit test, an icosahedron (and a dodecahedron) will appear in the output file `polyhedra.png`, thanks to the function `TestPolyhedra` in `main.cpp`. (See Figure 12.5.)

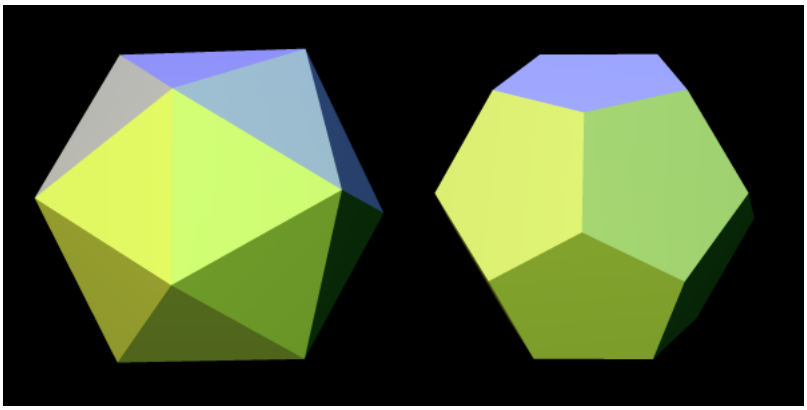

Figure 12.5: Ray-traced image of an icosahedron (left) and a dodecahedron (right).

Writing code for a complex solid with many triangular faces, as we see in the `Icosahedron` constructor, can be a bit challenging. I happened to have a paper model of an icosahedron already cut, folded, and glued for use as a reference. I wrote the numbers 0 to 11 on the 12 vertices of my paper model, and the letters "a" through "t" on its 20 faces. Each numbered vertex became a call to `AddPoint`, and each lettered face became a call to `AddTriangle`. As I coded each call to `AddTriangle`, I would be sure to type in its three vertices in a counterclockwise order, so that `TriangleMesh::AppendAllIntersections` would calculate surface normal vectors correctly pointing outward from the icosahedron's body (instead of inward). It may sound primitive and tedious to build a paper model like this, but I really don't know a better way to visualize the relationships between the vertices and triangular faces for a complicated solid like an icosahedron. I tried to draw diagrams on paper, but kept ending up with confusing messes. With the model in hand, it took just a few minutes to write the `Icosahedron` constructor, and it worked correctly the first time—no debugging needed! I should also mention that Wikipedia's article on the icosahedron was very helpful in that it explained the exact coordinates for all the vertex points. That made it very easy to code all the calls to `AddPoint`.

12.7 Using `TriangleMesh` for polygons other than triangles

`TriangleMesh` can be used to build solids that have faces that are polygons other than triangles. Any polygon can be split up into triangles. For example, quadrilaterals and pentagons can be replaced with equivalent triangles as shown in Figures 12.6 and 12.7.

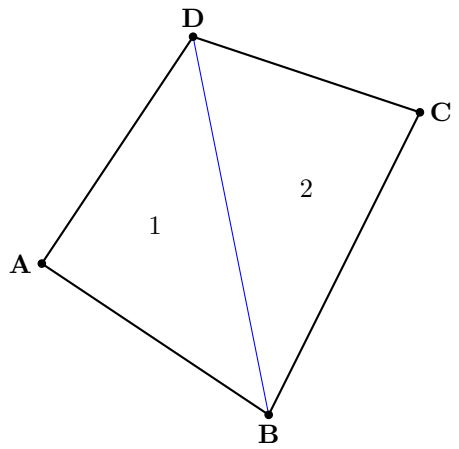

Figure 12.6: A quadrilateral can be split into 2 triangles.

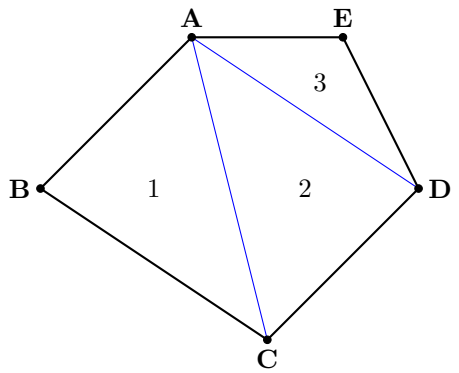

Figure 12.7: A pentagon can be split into 3 triangles.

`TriangleMesh` provides helper methods for these two cases called `AddQuad` and `AddPentagon`. As you can see in `imager.h`, these inline methods result in multiple calls to `AddTriangle`. Just like `AddTriangle`, these methods require the caller to pass the point indices in a counterclockwise order as seen from outside the solid object. It is possible to modify the code to add more helper methods for polygons with even more sides: hexagons, heptagons, etc., if you wish.

You can use `TriangleMesh` to model all kinds of complicated solid objects using polygonal faces. With some careful planning (perhaps with the help of some paper and glue before programming) you can bring these objects to life and place them in a scene, with realistic three-dimensional lighting and shadows.

Chapter 13

The `Torus` class

13.1 Mathematics of the torus

We now explore how to use the ray tracer code to draw a torus, which is more commonly known as a donut shape. This shape is implemented by `class Torus`, which is declared in `imager.h` and implemented in `torus.cpp`. Two constants suffice to define the shape of the torus: A, the distance from the center of the hole inside the torus and the center of its tube, and B, the radius of the tube, as shown in Figure 13.1. In order to ensure that there is a hole inside the torus, we will assume that $A > B$.

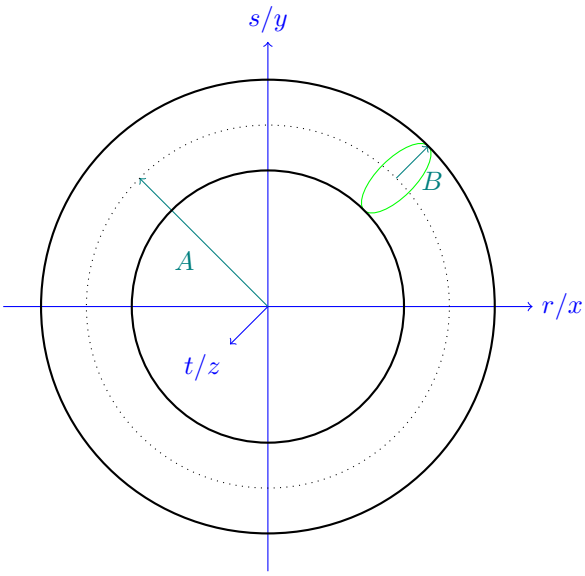

Figure 13.1: The torus dimensions A and B.

Note that `Torus` is a reorientable solid (it derives from `SolidObject_Reorientable`), so it technically resides in object space coordinates (r, s, t). However, I will use (x, y, z) throughout the derivation here to more closely match the `x`, `y`, and `z` member variables of the C++ type `Vector`. Hopefully this will be more helpful than confusing when looking back and forth between the C++ code and the math in this chapter.

Our torus is centered at the origin $(0, 0, 0)$ and oriented such that it is cut in half (like a bagel) by the xy plane. The following equation describes every point on the surface of a torus aligned this way. See the Wikipedia article *Torus* for more information about the derivation of this formula.

$$(x^2 + y^2 + z^2 + A^2 - B^2)^2 = 4A^2(x^2 + y^2)$$

13.2 Intersection of a ray and a torus

As with every other kind of solid object we want to depict with the ray tracer code, we will need to solve equations for intersections with a ray, and figure out how to calculate a surface normal vector for an intersection point once we find it. This will be the most challenging of the algebra problems we have faced so far. As usual, we will substitute occurrences of x, y, and z in the torus equation with linear expressions of the parameter u based on the parametric ray equation $\mathbf{P} = \mathbf{D} + u\mathbf{E}$:

$$\begin{cases} x = E_x u + D_x \\ y = E_y u + D_y \\ z = E_z u + D_z \end{cases}$$

The substitution yields a single equation in u that expresses where along the ray (if anywhere) that ray passes through the surface of the torus:

$$\begin{aligned} & \left[(E_x u + D_x)^2 + (E_y u + D_y)^2 + (E_z u + D_z)^2 + (A^2 - B^2)\right]^2 \\ = \quad & 4A^2 \left[(E_x u + D_x)^2 + (E_y u + D_y)^2\right] \end{aligned}$$

Expanding the innermost squared terms that involve u, we find

$$\begin{aligned} & \left[(E_x^2 u^2 + 2D_x E_x u + D_x^2) \right. \\ & +(E_y^2 u^2 + 2D_y E_y u + D_y^2) \\ & \left. +(E_z^2 u^2 + 2D_z E_z u + D_z^2)\right]^2 \\ & = \\ 4A^2 & \left[(E_x^2 u^2 + 2D_x E_x u + D_x^2) \right. \\ & \left. +(E_y^2 u^2 + 2D_y E_y u + D_y^2)\right] \end{aligned}$$

Collecting like powers of u on both sides of the equation, we have

$$\begin{aligned} & \left[(E_x^2 + E_y^2 + E_z^2)u^2 + 2(D_x E_x + D_y E_y + D_z E_z)u + (D_x^2 + D_y^2 + D_z^2 + A^2 - B^2)\right]^2 \\ = \quad & 4A^2 \left[(E_x^2 + E_y^2)u^2 + 2(D_x E_x + D_y E_y)u + (D_x^2 + D_y^2)\right] \end{aligned}$$

Let's define some constant symbols to simplify the algebra. Note that all of these definitions contain nothing but known constant terms, so the C++ algorithm will have no difficulty calculating their values.

$$\text{Let} \begin{cases} G & = 4A^2(E_x^2 + E_y^2) \\ H & = 8A^2(D_x E_x + D_y E_y) \\ I & = 4A^2(D_x^2 + D_y^2) \\ J & = E_x^2 + E_y^2 + E_z^2 = |\mathbf{E}|^2 \\ K & = 2(D_x E_x + D_y E_y + D_z E_z) = 2(\mathbf{D} \cdot \mathbf{E}) \\ L & = D_x^2 + D_y^2 + D_z^2 + A^2 - B^2 = |\mathbf{D}|^2 + (A^2 - B^2) \end{cases}$$

The equation becomes much more concise:

$$(Ju^2 + Ku + L)^2 = Gu^2 + Hu + I$$

Expanding the squared term on the left is aided by a 3-by-3 grid showing every term multiplied by every term, as shown in Figure 13.2.

Grouping the grid in upward sloping diagonals, as shown by the five curves in the figure, reveals collections of like powers of u. Summing the terms along these diagonals and factoring out the common powers of u, we have:

$$J^2 u^4 + 2JKu^3 + (2JL + K^2)u^2 + 2KLu + L^2 = Gu^2 + Hu + I$$

Subtracting the right side of the equation from both sides and collecting like terms, the equation becomes:

$$J^2 u^4 + 2JKu^3 + (2JL + K^2 - G)u^2 + (2KL - H)u + (L^2 - I) = 0$$

	Ju^2	Ku	L
Ju^2	J^2u^4	JKu^3	JLu^2
Ku	JKu^3	K^2u^2	KLu
L	JLu^2	KLu	L^2

Figure 13.2: Grid that aids in expanding $(Ju^2 + Ku + L)^2$.

13.3 Solving the quartic intersection equation

An equation like this, where the unknown variable u appears as u^4, u^3, u^2, u^1, and u^0 (the constant term $L^2 - I^2$) is known as a *quartic equation*. It is solvable analytically, but the procedure is very complicated. I will not detail the solution here because there are many resources on the Internet that do a fine job. The source file `algebra.cpp` and its header file `algebra.h` contain a pair of overloaded functions called `SolveQuarticEquation`, one geared for quartic equations with all real coefficients (as we have here), and another more general version for those with complex coefficients. The complex version of `SolveQuarticEquation` is based on techniques from the Wikipedia article called *Quartic equation*. The real version calls the complex version and filters out any solutions that are not real numbers. Only real values of u have meaning in the parametric ray equation, so we use the real version of `SolveQuadraticEquation` to solve the torus equation above.

Take a look in `torus.cpp` at the method `Torus::SolveIntersections`. Given a vantage point and a direction vector, it finds all positive real solutions for u. Because the intersection equation is quartic in u, there can be up to four real solutions. This makes sense intuitively because a ray can pierce a torus in up to four places, marked u_0 through u_3 in Figure 13.3.

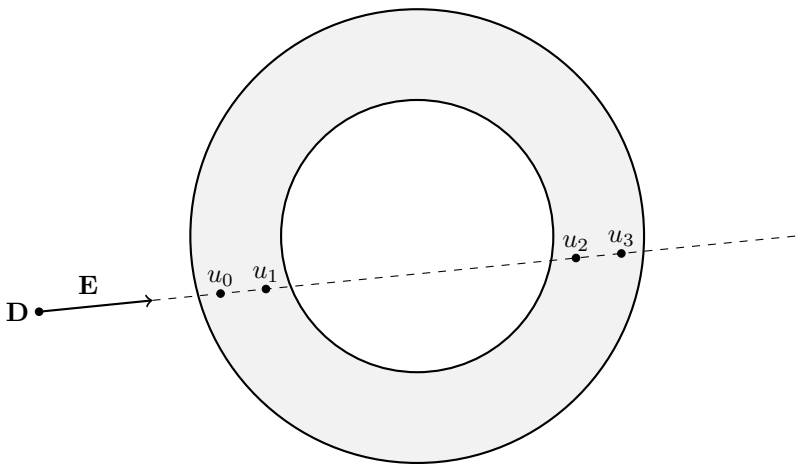

Figure 13.3: A ray can pierce a torus in up to four locations.

`Torus::SolveIntersections` fills in the parameter `uArray` with 0 to 4 positive real solutions for u, and returns the number of solutions. The caller must use the return value to know how many of the elements of `uArray` are valid.

The `Torus` class derives from `SolidObject_Reorientable` so that the math can assume a fixed position and orientation in space, while still allowing us to create images of a torus translated and rotated any way we want. `Torus::ObjectSpace_AppendAllIntersections` receives a vantage point and direction vector that have already been translated into object space coordinates. It calls `SolveIntersections` to find 0 to 4 positive real values of u. Each valid value u value gives us an intersection point when u is plugged into the parametric ray equation $\mathbf{P} = \mathbf{D} + u\mathbf{E}$.

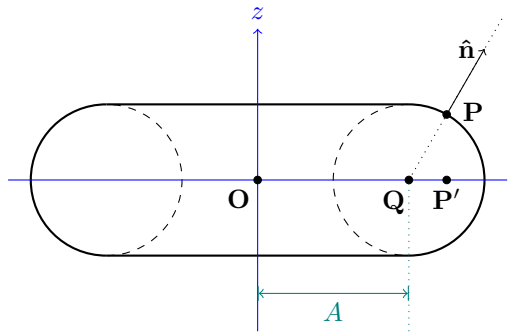

Figure 13.4: Surface normal vector $\hat{\mathbf{n}}$ for a point on a torus (edge view).

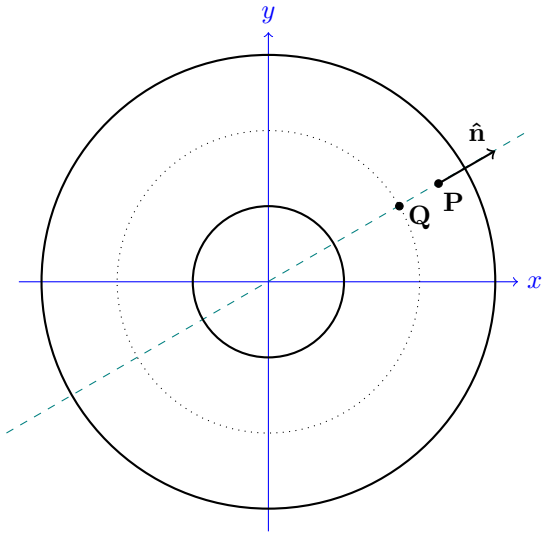

Figure 13.5: Surface normal vector $\hat{\mathbf{n}}$ for a point on a torus (top view).

13.4 Surface normal vector for a point on a torus

We are left with the problem of finding the surface normal unit vector for a point on the surface of a torus. It is possible to solve this problem using 3D calculus and partial derivatives, but with a little geometric intuition and a pair of diagrams, there is a much easier way. Figures 13.4 and 13.5 show two views of a point $\mathbf{P}$ on the surface of a torus, one edge-on, the other top-down.

In both views, $\mathbf{O}$ is the origin $(0, 0, 0)$, which is also the center of the torus hole. We let $\mathbf{Q}$ be the point at the center of the torus tube that is closest to the intersection point $\mathbf{P}$. The edge-on view on the left shows a cross-section of the torus passing through the plane of $\mathbf{O}$, $\mathbf{P}$, and $\mathbf{Q}$. The top-down view on the right depicts that same cross-section plane as a diagonal dashed line. Looking at these diagrams, it becomes clear that the surface normal vector points in the exact opposite direction from $\mathbf{P}$ as the point $\mathbf{Q}$ does.

Assuming we have already calculated $\mathbf{P}$ from u using $\mathbf{P} = \mathbf{D} + u\mathbf{E}$, how do we determine where $\mathbf{Q}$ is? The answer comes from combining three facts we know about $\mathbf{Q}$:

1. $\mathbf{Q}$ lies on the xy plane, so its z component must be 0. This means $\mathbf{Q} = (Q_x, Q_y, 0)$, where Q_x and Q_y are unknowns to be found.

2. The distance from $\mathbf{Q}$ to the origin $\mathbf{O}$ is the constant A (by definition—A is the distance from the center of the torus hole to any point on the center of the solid tube). So $|\mathbf{Q}| = A$.

3. The "shadow" of $\mathbf{P}$ on the xy plane (marked $\mathbf{P}'$ in Figure 13.4) lies in the same direction from the origin as $\mathbf{Q}$. If $\mathbf{P} = (P_x, P_y, P_z)$, then $\mathbf{P}' = (P_x, P_y, 0)$.

We find $\mathbf{Q}$ by converting $\mathbf{P}'$ into a unit vector, then multiplying that unit vector by A (which

we know to be the same as $|\mathbf{Q}|$), giving us $\mathbf{Q}$ itself:

$$\mathbf{Q} = A\frac{\mathbf{P}'}{|\mathbf{P}'|} = \left(\frac{A}{|\mathbf{P}'|}\right)\mathbf{P}'$$
$$\mathbf{Q} = \frac{A}{\sqrt{P_x^2 + P_y^2}}(P_x, P_y, 0)$$

Let $\mathbf{N}$ be the vector from $\mathbf{Q}$ to $\mathbf{P}$:

$$\mathbf{N} = \mathbf{P} - \mathbf{Q}$$
$$\mathbf{N} = (P_x, P_y, P_z) - \frac{R}{\sqrt{P_x^2 + P_y^2}}(P_x, P_y, 0)$$

To make the formula simpler, we can let

$$\alpha = \frac{R}{\sqrt{P_x^2 + P_y^2}}$$

which gives us

$$\mathbf{N} = (P_x, P_y, P_z) - (\alpha P_x, \alpha P_y, 0)$$
$$\mathbf{N} = ((1-\alpha)P_x, (1-\alpha)P_y, P_z)$$

Converting $\mathbf{N}$ to a unit vector is simply a matter of dividing $\mathbf{N}$ by its own magnitude, resulting in the surface normal unit vector $\hat{\mathbf{n}}$:

$$\hat{\mathbf{n}} = \frac{\mathbf{N}}{|\mathbf{N}|}$$

The member function `Torus::SurfaceNormal` in `torus.cpp` performs this calculation for any point $\mathbf{P}$ on the surface of a torus.

Using the algorithms explained above, the member function `ObjectSpace_AppendAllIntersections` in the class `Torus` finds intersections with a ray and the torus, and for each intersection point, determines its surface normal unit vector. As with every other solid object, these two vectors and the color of the surface are all the ray tracer needs to draw an image.

Chapter 14

Set Operations

14.1 A simpler way to code certain shapes

Although it is possible to create a custom class for every desired shape one might imagine, it does involve a fair amount of mathematical and programming effort each time. Some shapes might be a combination of shapes we have already implemented, or a portion of an existing shape. For cases like these, it is often possible to avoid a lot of effort by using some C++ classes in `imager.h` that allow us to combine other shapes to create new ones. These classes are called *set operators*. We can think of any solid object as a set of points in space. For example, a sphere is the set of all points whose distance from its center is less than or equal to its radius. The set operator classes are:

- `SetUnion`
- `SetIntersection`
- `SetComplement`
- `SetDifference`

14.2 `SetUnion`

This is the simplest set operator to understand. Given two `SolidObject` instances, the `SetUnion` creates a single object that includes both of them. For example, if you had code that created a sphere and a cube, the union of the sphere and the cube would act as a single object comprised of both objects together. The sphere and cube might overlap or they might have some empty space between them. But in either case, you could rotate or translate the set union, and the sphere and cube would both respond to each rotation or translation. But `class SetUnion` is more than a mere container; its power increases when used in combination with other set operators, as we will see later.

14.3 `SetIntersection`

Like `SetUnion`, `class SetIntersection` operates on two solid objects to produce a new solid object. However, instead of including all points that are in either of the solid objects like `SetUnion` does, `SetIntersection` includes only the points that are in **both** of the objects. Using `SetIntersection` makes sense only when the two objects overlap to some extent. For example, let's take a look at what happens when we create two overlapping spheres and feed them to `SetIntersection`. See Figure 14.1.

The resulting solid is not a sphere, but a sort of lens shape with a sharp circular edge. The function `SetIntersectionTest` in `main.cpp` shows exactly how to implement this overlapping sphere intersection example. If you have run the unit tests, it has already created the image file `intersection.png`, reproduced in Figure 14.2.

It may seem odd that the left and right parts of the lens are shaded differently. If you look at the image file `intersection.png` you will see that the left part of the lens is purple and the

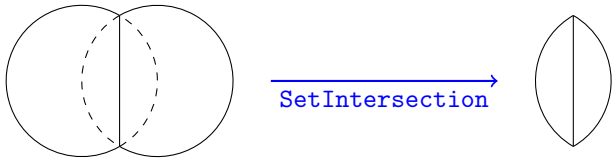

Figure 14.1: The set intersection of two overlapping spheres.

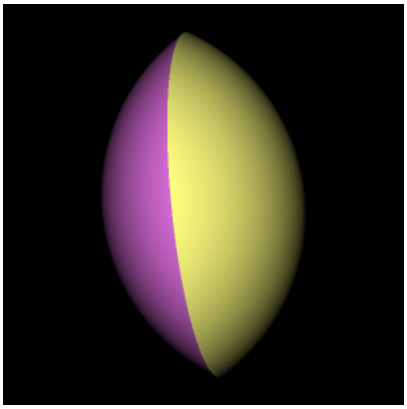

Figure 14.2: Ray-traced image showing intersection of two spheres.

right part is yellow. Let's take a closer look in `main.cpp` at the source code that creates the two spheres and their set intersection (the function `SetIntersectionTest`), to understand the coloring oddity:

```
// Display the intersection of two overlapping spheres.
const double radius = 1.0;

Sphere* sphere1 = new Sphere(
    Vector(-0.5, 0.0, 0.0),
    radius,
    Color(1.0, 1.0, 0.5)
);

Sphere* sphere2 = new Sphere(
    Vector(+0.5, 0.0, 0.0),
    radius,
    Color(1.0, 0.5, 1.0)
);

SetIntersection *isect = new SetIntersection(
    Vector(0.0, 0.0, 0.0),
    sphere1,
    sphere2
);
```

The color parameter used to create `sphere1`, `Color(1.0, 1.0, 0.5)`, has more red and green, but less blue, resulting in an overall yellowish tone. On the other hand, `sphere2`'s color is more red and blue than yellow, giving it a purplish tone. The rule for a set intersection's surface color is that when a camera ray intersects with one of the two solids, the intersection point takes on that solid's surface color if the same point is contained by the other solid. (If not contained by the second solid, the point is ignored because the two solids do not overlap at that location.) The rationale for this rule is that only the surfaces of a solid (not its insides) have a color, and each class derived from `SolidObject` is free to assign colors to every point on its surface as it sees fit—there is no requirement that a solid's surface be of a uniform color. If you, as an artist/programmer, wish to create a set intersection object with a consistent color across its entire surface, you will need to give the `SetIntersection` constructor two objects of the same color.

14.4 The SolidObject::Contains member function

At this time we need to rivisit more thoroughly a function that was mentioned earlier, but only in passing: the virtual method SolidObject::Contains. Take a look in imager.h at the class SolidObject declaration. You will see the following within it:

```
// Returns true if the given point is inside this solid object.
// This is a default implementation that counts intersections
// that enter or exit the solid in a given direction from the point.
// Derived classes can often implement a more efficient algorithm
// to override this default algorithm.
virtual bool Contains(const Vector& point) const;
```

This member function is implemented in the source file solid.cpp. This default implementation will work for any solid that is fully enclosed, meaning that its surfaces completely seal off an interior volume of space without leaving any cracks or gaps. By default, a SolidObject is assumed to be fully enclosed, but any derived class can override this behavior by passing false for the _isFullyEnclosed parameter of the SolidObject constructor. In this case, SolidObject::Contains will immediately return false every time it is called. Such an object will not be able to participate in set intersection operations (or refraction for that matter).

However, if the solid object is created claiming to be fully enclosed via passing the default value of the _isFullyEnclosed parameter (true), then SolidObject::Contains will determine whether a point is contained by the solid using a boundary-crossing algorithm: the function draws a ray from the point in question and counts how many times the ray enters or leaves the solid. It does this using the same AppendAllIntersections function used by the ray tracer to follow rays of light through the scene. If a point is inside a fully-enclosed solid, no matter what the shape of that solid is, a ray will exit the solid one more time than it enters the solid. For example, starting at a point inside a sphere, the ray would intersect with the sphere one time, resulting in a single exit point and no entry points. If a point is outside the solid, the number of entry and exit points is the same, no matter which direction you draw the ray.

This algorithm works fine, but it is somewhat complicated and not ideal in its run-time efficiency. Therefore, wherever possible, it is a good idea to override Contains on any new classes you derive from SolidObject. If your class derives from SolidObject_Reorientable, you should override ObjectSpace_Contains instead, which does the same thing as Contains, only it is passed a point that has already been converted from camera space coordinates to object space coordinates. For example, class Sphere implements Contains, while class Torus implements ObjectSpace_Contains. Of all the solid object classes I wrote for this book, only TriangleMesh uses the default implementation of Contains; in every other case, it was easy to create a much more efficient replacement.

Whenever SetIntersection finds a ray intersecting with one of its two inner solids, it calls the Contains method on the other solid to determine whether that point is part of the common volume occupied by both objects (Contains returns true) or whether the point should be ignored (Contains returns false).

Incidentally, you may remember from an earlier chapter that the Contains method is also called by the refraction code to determine the refractive index of an arbitrary point in space: the scene iterates through the solid objects inside it, asking each of them if it contains the point. If a point is claimed by a solid, that solid's refractive index is used as the refractive index of that point in space.

14.5 SetComplement

Unlike SetUnion and SetIntersection, which operate on two solids, class SetComplement operates on a single solid. The complement of a solid object is the set of points that are **not** inside that object. If a point is contained by a solid, it is not contained by the set's complement, and if a point is not contained by a solid, it **is** contained by the complement. As an example of this idea, imagine an instance of class Sphere. Ray-tracing an image of it results in what looks like a ball of solid matter floating in an infinite void. The complement of this sphere would be an infinite expanse of solid matter in all directions, with a spherical hole inside it. This does not sound very useful—the camera immersed in an infinite expanse of opaque substance will not

be able to see anything! The truth is, `SetComplement` used by itself does not make much sense, but it is very useful when combined with the other set operators, as we shall soon see.

As you might expect, when `SetComplement::Contains` is called, it calls the `Contains` method on its inner object and returns the opposite boolean value. Returning to the same example, if the inner object is a sphere, and `Sphere::Contains` reports that a given point is inside the sphere by returning `true`, `SetComplement::Contains` returns `false`, indicating that the point resides within the spherical bubble, i.e., outside the solid matter that comprises the complement.

Interestingly, `SetComplement::AppendAllIntersections` simply calls the `AppendAllIntersections` method on its inner object, because an object's surfaces are in exactly the same places as those of the object's complement. However, it must modify the surface normal unit vector for each `Intersection` struct reported by the inner object, because the complement operation effectively turns the inner object inside-out. A sphere with normal vectors pointing outward from the sphere's solid matter into empty space becomes an empty bubble with normal vectors pointing inward, again away from the infinite expanse of solid matter and into the empty space within the bubble. These implementation details are crucial to make `class SetComplement` behave correctly when combined with the other set operators.

14.6 `SetDifference`

Like `SetUnion` and `SetIntersection`, `SetDifference` operates on a pair of inner objects. Unlike those other binary set operators, `SetDifference` means different things depending on the order the two inner objects are passed into its constructor. We will call the first inner object the *left* object and the second object the *right* object. The `SetDifference` creates a solid object that contains all the points of the left object **except** those that belong to the right object. Figuratively speaking, the left object is like a piece of stone, and the right object (to the extent that it overlaps in space with the left object) determines which parts of the left object should be carved away.

Like `SetIntersection`, `SetDifference` makes sense only when the right object overlaps with the left object at least a little bit. If the two inner objects of a `SetIntersection` nowhere overlap, the result is no visible object at all—just complete emptiness. Non-overlapping inner objects fed to `SetDifference` result in an object that looks just like the left object. In both cases, the set operator adds no value, and just slows down execution of the ray tracer with useless work. The algorithm will still work correctly in a mathematical sense, but with no practical benefit.

`SetDifference` is implemented using `SetIntersection` and `SetComplement`. The difference of the left object `L` and the right object `R` is defined as the intersection of `L` and the complement of `R`:

```
difference(L, R) = intersection(L, complement(R))
```

We can read this as saying that a point is in the set difference only if it is inside the left object but **not** inside the right object.

The ray tracer unit test exercises the `SetDifference` operator with the function `SetDifferenceTest`, located in `main.cpp`. This test function creates a `SetDifference` object having a torus as its left object and a sphere as its right object. The resulting image looks like a donut with a spherical bite taken out of it, hence the image file's name, `donutbite.png`. See Figure 14.3.

14.7 Combining set operations in interesting ways

Set operators would be quite limited if their operands could only be simple shapes. Fortunately for our creative freedom, the classes `SetUnion`, `SetIntersection`, `SetComplement`, and `SetDifference` derive from `SolidObject`, so we can pass set operators as constructor arguments to other set operators. One use of this technique is to circumvent the apparent limitation of `SetUnion`, `SetIntersection`, or `SetDifference` operating on only two objects. Suppose we want to create the set intersection of three objects `A`, `B`, and `C`. Because the `SetIntersection` constructor allows only two `SolidObject` instances to be passed as arguments, we cannot use it to directly represent `intersection(A,B,C)`. However, we can create an object `intersection(A,B)`,

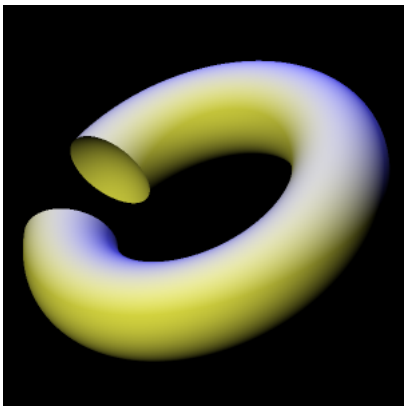

Figure 14.3: Torus with a sphere subtracted from it, using `SetDifference`.

and pass that new object as the left argument, along with `C` as the right argument, to create another intersection object. Conceptually, this looks like the following pseudo-code:

```
intersection(intersection(A, B), C)
```

In actual C++ code, we must also pass the location in space that serves as the center of the object formed by the set operator. The center is represented as a vector, and all rotations of the set operator object will cause it to pivot about that center point. Also, all inner objects must be created using dynamic allocation via `operator new`, or a crash will occur in the set operator's destructor when it attempts to use `operator delete` to destroy its inner objects. So with these details in mind, here is a more realistic listing of a set intersection applied to three objects `A`, `B`, and `C`:

```
SolidObject* A = new /*Whatever*/;
SolidObject* B = new /*Whatever*/;
SolidObject* C = new /*Whatever*/;
SolidObject* isect = new SetIntersection(
    A->Center(),
    new SetIntersection(A->Center(), A, B),
    C
);
scene.AddSolidObject(isect);
```

This listing shows `A`'s center point used as the center for the intersection object's center, but any other point is allowed; you are free to choose any point in space for the intersection object to revolve about. Also, if you use the `Move` member function to move the intersection object to another location, the existing center point will be moved to that new location, and all inner objects (`A`, `B`, and `C` in this case) will be shifted by the same amount in the x, y, and z directions. For example, if `A->Center()` and `isect->Center()` are both $(1, 2, 3)$, `B->Center()` is $(5, 10, 4)$, and `C->Center()` is $(0, 6, 2)$, and we do `isect->Move(1,1,1)`, then all three objects will by moved by the amount $(1, 1, 1) - (1, 2, 3) = (0, -1, -2)$. So `A` and `isect` will both end up centered at $(5, 9, 2)$ and `C` will be centered at $(0, 5, 0)$.

14.8 A set operator example: concrete block

Now we will explore how to use set operators to create an image of a concrete cinder block as shown in Figure 14.4.

In Figure 14.5 we see that the concrete block shape consists of the large cuboid shown on the left with two smaller cuboids on the right subtracted from it.

The header file `block.h` shows how to create the concrete block shape with a concise class declaration. The class `ConcreteBlock` derives from `SetDifference`. It creates a large cuboid as the `SetDifference`'s left object, and a `SetUnion` of the two smaller cuboids as the `SetDifference`'s right object. Referring to th labels `A`, `B`, and `C` in the previous diagram, we have the `ConcreteBlock` defined conceptually as

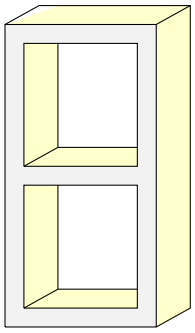

Figure 14.4: The shape represented by `class ConcreteBlock`.

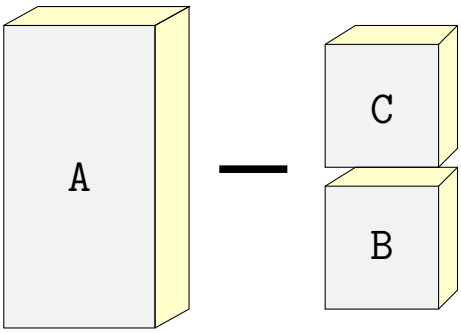

Figure 14.5: Subtracting the small cuboids from the large cuboid generates the concrete block.

```
block = difference(A, union(B, C))
```

The ray tracer unit test calls the function `BlockTest` in `main.cpp`, and that function creates an image of a `ConcreteBlock` with a `Sphere` casting a shadow onto it. The output file is `block.png`, which is reproduced in Figure 14.6.

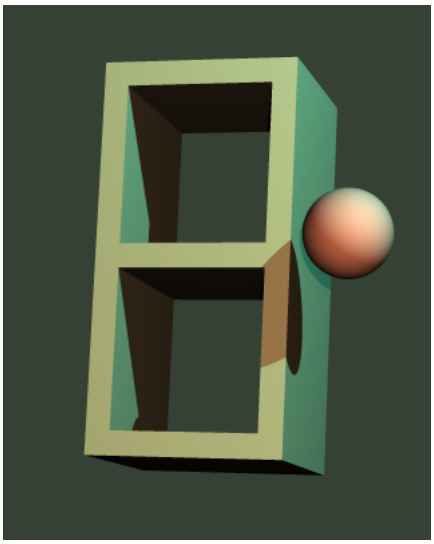

Figure 14.6: Concrete block image created by the function `BlockTest` in `main.cpp`.