(for ECE660, Fall, 1999)

Chapter 4. Vectors Tools for Graphics.

"The knowledge at which geometry aims is knowledge of the eternal,
and not of aught perishing and transient."
Plato

For us, whose shoulders sag under the weight of the heritage of Greek thought
and who walk in the paths traced out by the heroes of the Renaissance,
a civilization without mathematics is unthinkable.
Andre Weil

"Let us grant that the pursuit of mathematics is a divine madness of the human spirit."
Alfred North Whitehead

"All that transcend geometry, transcends our comprehension".
Blaise Pascal

Goals of the Chapter

- To review vector arithmetic, and to relate vectors to objects of interest in graphics.
- To relate geometric concepts to their algebraic representations.
- To describe lines and planes parametrically.
- To distinguish points and vectors properly.
- To exploit the dot product in graphics topics.
- To develop tools for working with objects in 3D space, including the cross product of two vectors.

Preview

This chapter develops a number of useful tools for dealing with geometric objects encountered in computer graphics. Section 4.1 motivates the use of vectors in graphics, and describes the principal coordinate systems used. Section 4.2 reviews the basic ideas of vectors, and describes the key operations that vectors allow. Although most results apply to any number of dimensions, vectors in 2D and 3D are stressed. Section 4.3 reviews the powerful dot product operation, and applies it to a number of geometric tasks, such as performing orthogonal projections, finding the distance from a point to a line, and finding the direction of a ray "reflected" from a shiny surface. Section 4.4 reviews the cross product of two vectors, and discusses its important applications in 3D graphics.

Section 4.5 introduces the notion of a coordinate frame and homogeneous coordinates, and stresses that points and vectors are significantly different types of geometric objects. It also develops the two principal mathematical representations of a line and a plane, and shows where each is useful. It also introduces affine combinations of points and describes an interesting kind of animation known as "tweening". A preview of Bezier curves is described as an application of tweening.

Section 4.6 examines the central problem of finding where two line segments intersect, which is vastly simplified by using vectors. It also discusses the problem of finding the unique circle determined by three points. Section 4.7 discusses the problem of finding where a "ray" hits a line or plane, and applies the notions to the clipping problem. Section 4.8 focuses on clipping lines against convex polygons and polyhedra, and develops the powerful Cyrus-Beck clipping algorithm.

The chapter ends with Case Studies that extend these tools and provide opportunities to enrich your graphics programming skills. Tasks include processing polygons, performing experiments in 2D "ray tracing", drawing rounded corners on figures, animation by tweening, and developing advanced clipping tools.

4.1 Introduction.

In computer graphics we work, of course, with objects defined in a three dimensional world (with 2D objects and worlds being just special cases). All objects to be drawn, and the "cameras" used to draw them, have shape, position, and orientation. We must write computer programs that somehow describe these objects, and describe how light bounces around illuminating them, so that the final pixel values on the display can be computed. Think of an animation where a camera flies through a hilly scene containing various buildings, trees, roads, and cars. What does the camera "see"? It all has to be converted ultimately to numbers. It's a tall order.

The two fundamental sets of tools that come to our aid in graphics are *vector analysis* and *transformations*. By studying them in detail we develop methods to describe the various geometric objects we will encounter, and we learn how to convert geometric ideas to numbers. This leads to a collection of crucial algorithms that we can call upon in graphics programs.

In this chapter we examine the fundamental operations of vector algebra, and see how they are used in graphics; transformations are addressed in Chapter 5. We start at the beginning and develop a number of important tools and methods of attack that will appear again and again throughout the book. If you have previously studied vectors much of this chapter will be familiar, but the numerous applications of vector analysis to geometric situations should still be scrutinized. The chapter might strike you as a mathematics text. But having it all collected in one place, and related to the real problems we encounter in graphics, may be found useful.

Why are vectors so important?
A preview of some of some situations where vector analysis comes to the rescue might help to motivate the study of vectors. Figure 4.1 shows three geometric problems that arise in graphics. Many other examples could be given.

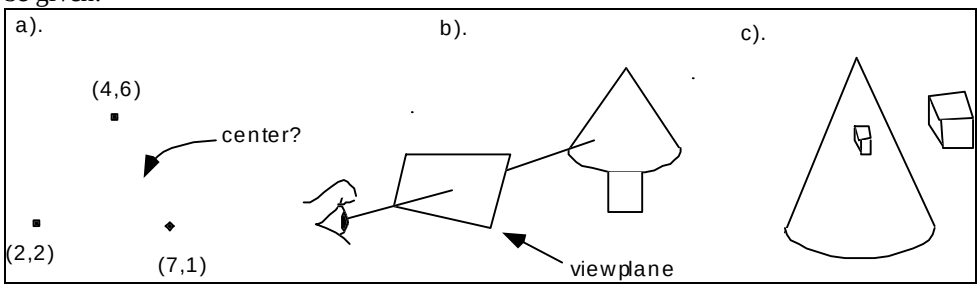


Figure 4.1. Three sample geometric problems that yield readily to vector analysis.

Part a) shows a computer-aided design problem: the user has placed three points on the display with the mouse, and wants to draw the unique circle that passes through them. (Can you visualize this circle?). For the coordinates given where is the center of the circle located? We see in Section 4.6 that this problem is thorny without the use of vectors, but almost trivial when the right vector tools are used.

Part b) shows a camera situated in a scene that contains a Christmas tree. The camera must form an image of the tree on its "viewplane" (similar to the film plane of a physical camera), which will be transferred to a screen window on the user's display. Where does the image of the tree appear on this plane, and what is its exact shape? To answer this we need a detailed study of perspective projections, which will be greatly aided by the use of vector tools. (If this seems too easy, imagine that you are developing an animation, and the camera is zooming in on the sphere along some trajectory, and rotating as it does so. Write a routine that generates the whole sequence of images!)

Part c) shows a shiny cone in which the reflection of a cube can be seen. Given the positions of the cone, cube, and viewing camera, where *exactly* does the reflected image appear, and what is its color and shape? When studying ray tracing in Chapter 15 we will make extensive use of vectors, and we will see that this problem is readily solved.

Some Basics.
All points and vectors we work with are defined relative to some coordinate system. Figure 4.2 shows the coordinate systems that are normally used. Each system has an *origin* called ϑ and some axes emanating from ϑ. The axes are usually oriented at right angles to one another. Distances are marked along each axis, and a

point is given coordinates according to how far along each axis it lies. Part a) shows the usual two-dimensional system. Part b) shows a *right handed* 3D coordinate system, and part c) shows a *left handed* coordinate system.

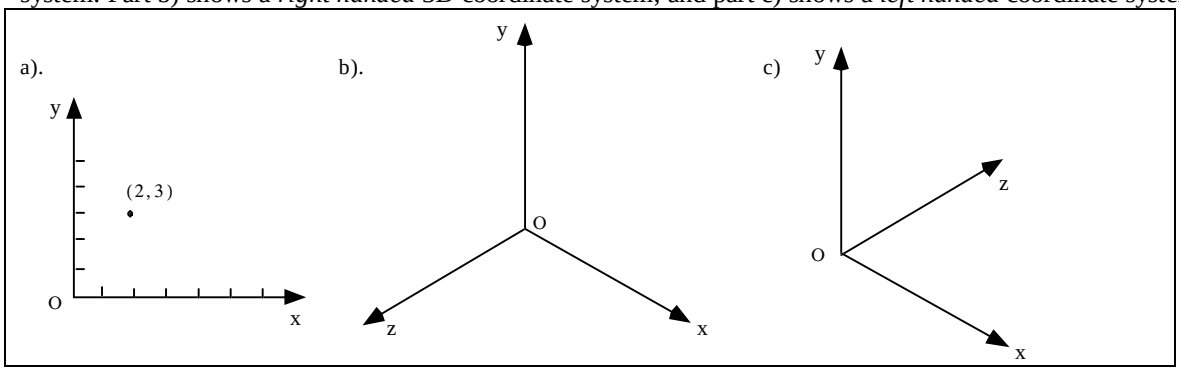


Figure 4.2. The familiar two- and three-dimensional coordinate systems.

In a right handed system, if you rotate your *right* hand around the *z*-axis by sweeping from the positive *x*-axis around to the positive *y*-axis, as shown in the figure, your thumb points along the positive *z*-axis. In a left handed system, you must do this with your *left* hand to make your thumb point along the positive *z*-axis. Right-handed systems are more familiar and are conventionally used in mathematics, physics, and engineering discussions. In this text we use a right-handed system when setting up models for objects. But left-handed systems also have a natural place in graphics, when dealing with viewing systems and "cameras".

We first look at the basics of vectors, how one works with them, and how they are useful in graphics. In Section 4.5 we return to fundamentals and show an important distinction between points and vectors that, if ignored, can cause great difficulties in graphics programs.

4.2. Review of Vectors.

"Not only Newton's laws, but also the other laws of physics, so far as we know today, have the two properties which we call invariance under translation of axes and rotation of axes. These properties are so important that a mathematical technique has been developed to take advantage of them in writing and using physical laws.. called vector analysis."

Richard Feynman

Vector arithmetic provides a unified way to express geometric ideas algebraically. In graphics we work with vectors of two, three, and four dimensions, but many results need only be stated once and they apply to vectors of any dimension. This makes it possible to bring together the various cases that arise in graphics together into a single expression, which can be applied to a broad variety of tasks.

Viewed geometrically, vectors are objects having length and direction. They correspond to various physical entities such as force, displacement, and velocity. A vector is often drawn as an arrow of a certain length pointing in a certain direction. It is valuable to think of a vector geometrically as a *displacement* from one point to another**.**

Figure 4.3 uses vectors to show how the stars in the Big Dipper are moving over time [kerr79]. The current location of each star is shown by a point, and a vector shows the velocity of each star. The "tip" of each arrow shows the point where its star will be located in 50,000 years: producing a very different Big Dipper indeed!

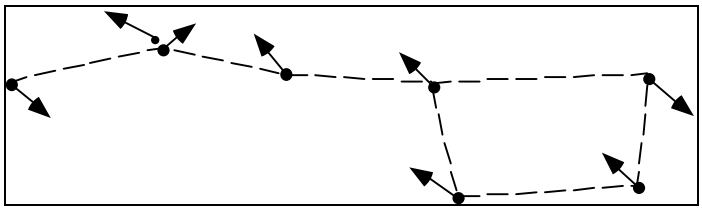


Figure 4.3. The Big Dipper now and in AD 50,000.

Figure 4.4a shows, in a 2D coordinate system, the two points $P = (1, 3)$ and $Q = (4, 1)$. The displacement from P to Q is a vector $\mathbf{v}$ having components $(3, -2)$,[1] calculated by subtracting the coordinates of the points individually. To "get from" P to Q we shift down by 2 and to the right by 3. Because a vector is a displacement it has size and direction but no inherent location: the two arrows labeled $\mathbf{v}$ in the figure are in fact the same vector. Figure 4.4b shows the corresponding situation in three dimensions: $\mathbf{v}$ is the vector from point P to point Q. One often states:

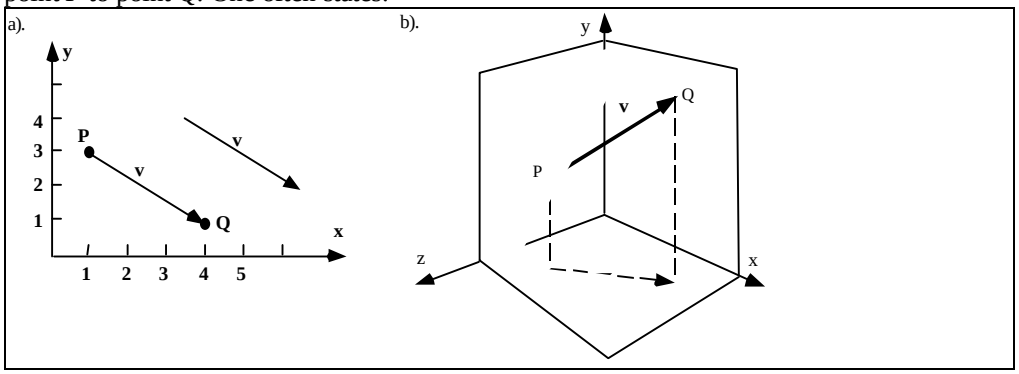


Figure 4.4. A vector as a displacement.

• The **difference** between two points is a vector: $\mathbf{v} = Q - P$;

Turning this around, we also say that a point Q is formed by displacing point P by vector $\mathbf{v}$; we say that $\mathbf{v}$ "offsets" P to form Q. Algebraically, Q is then the **sum**: $Q = P + \mathbf{v}$.

• The **sum** of a point and a vector is a point: $P + \mathbf{v} = Q$.

At this point we represent a vector through a list of its components: an n-dimensional vector is given by an *n-tuple*:

$$\mathbf{w} = (w_1, w_2, \ldots, w_n) \qquad (4.1)$$

Mostly we will be interested in 2D or 3D vectors as in $\mathbf{r} = (3.4, -7.78)$ or $\mathbf{t} = (33, 142.7, 89.1)$. Later when it becomes important we will explore the distinction between a vector and its *representation*, and in fact will use a slightly expanded notation to represent vectors (and points). Writing a vector as a *row matrix* like $\mathbf{t} = (33, 142.7, 89.1)$ fits nicely on the page, but when it matters we will instead write vectors as *column matrices*:

$$\mathbf{r} = \begin{pmatrix} 3.4 \\ -7.78 \end{pmatrix}, \text{ or } \mathbf{t} = \begin{pmatrix} 33 \\ 142.7 \\ 89.1 \end{pmatrix}$$

It matters when we want to multiply a point or a vector by a matrix, as we shall see in Chapter 5.

4.2.1. Operations with vectors.

Vectors permit two fundamental operations: you can add them, and you can multiply them by **scalars** (real numbers)[2]. So if $\mathbf{a}$ and $\mathbf{b}$ are two vectors, and s is a scalar, it is meaningful to form both $\mathbf{a} + \mathbf{b}$ and the product $s\mathbf{a}$. For example, if $\mathbf{a} = (2, 5, 6)$ and $\mathbf{b} = (-2, 7, 1)$, we can form the two vectors:

$\mathbf{a} + \mathbf{b} = (0, 12, 7)$

[1] Upper case letters are conventionally used for points, and boldface lower case letters for vectors.

[2] There are also systems where the scalars can be complex numbers; we do not work with them here.

6 **a** = (12, 30, 36)

always performing the operations *componentwise*. Figure 4.5 shows a two-dimensional example, using **a** = (1, -1) and **b** = (2, 1). We can represent the addition of two vectors graphically in two different ways. In Figure 4.5a we show both vectors "starting" at the same point, thereby forming two sides of a parallelogram. The sum of the vectors is then a diagonal of this parallelogram, the diagonal that emanates from the binding point of the vectors. This view — the "parallelogram rule" for adding vectors — is the natural picture for forces acting at a point: The diagonal gives the resultant force.

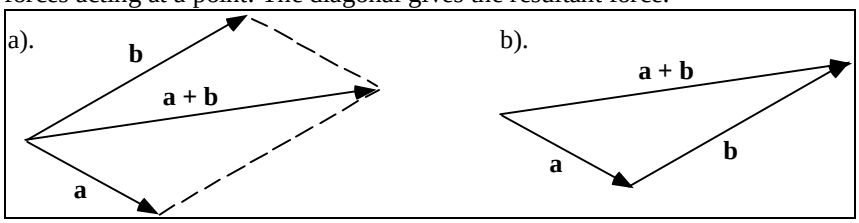


Figure 4.5. The sum of two vectors.

Alternatively, in Figure 4.5b we show one vector starting at the head of the other (i.e., place the tail of **b** at the head of **a**) and draw the sum as emanating from the tail of **a** to the head of **b**. The sum completes the triangle, which is the simple addition of one displacement to another. The components of the sum are clearly the sums of the components of its parts, as the algebra dictates.

Figure 4.6 shows the effect of scaling a vector. For $s = 2.5$ the vector s **a** has the same direction as **a** but is 2.5 times as long. When s is negative, the direction of s **a** is opposite that of **a**: The case $s = -1$ is shown in the figure.

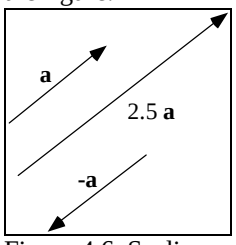


Figure 4.6. Scaling a vector.

Subtraction follows easily once adding and scaling have been established: **a** - **c** is simply **a** + (-**c**). Figure 4.7 shows the geometric interpretation of this operation, forming the difference of **a** and **c** as the sum of **a** and -**c** (Figure 4.7b). Using the parallelogram rule, this sum is seen to be equal to the vector that

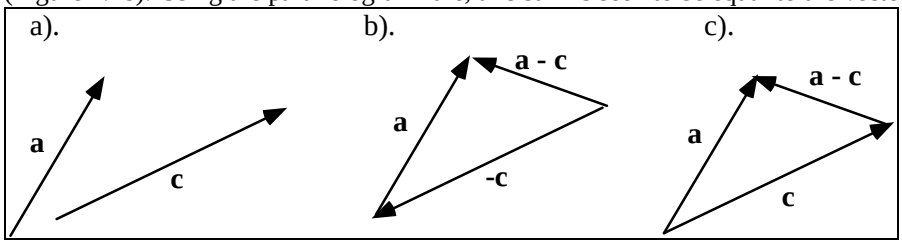


Figure 4.7. Subtracting vectors.

emanates from the head of **c** and terminates at the head of **a** (Figure 4.7c). This is recognized as one diagonal of the parallelogram constructed using **a** and **c**. Note too that it is the "other" diagonal from the one that represents the sum **a** + **c**.

4.2.2. Linear Combinations of Vectors.

With methods in hand for adding and scaling vectors, we can define a linear combination of vectors. To form a **linear combination** of two vectors, **v** and **w**, (having the same dimension) we scale each of them by some scalars, say a and b, and add the weighted versions to form the new vector, a **v** + b **w**. The more general definition for combining m such vectors is:

Definition:
A **linear combination** of the *m* vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m$ is a vector of the form

$$\mathbf{w} = a_1 \mathbf{v}_1 + a_2 \mathbf{v}_2 + \ldots + a_m \mathbf{v}_m \qquad (4.2)$$

where $a_1, a_2, \ldots, a_m$ are scalars.

For example, the linear combination 2(3, 4,-1) + 6(-1, 0, 2) forms the vector (0, 8, 10). In later chapters we shall deal with rather elaborate linear combinations of vectors, especially when representing curves and surfaces using spline functions.

Two special types of linear combinations, "affine" and "convex" combinations, are particularly important in graphics.

Affine Combinations of Vectors.

A linear combination is an **affine combination** if the coefficients $a_1, a_2, \ldots, a_m$ add up to 1. Thus the linear combination in Equation 4.2 is affine if:

$$a_1 + a_2 + \ldots + a_m = 1 \qquad (4.3)$$

For example, 3 **a** + 2 **b** - 4 **c** is an affine combination of **a**, **b**, and **c**, but 3 **a** + **b** - 4 **c** is not. The coefficients of an affine combination of two vectors **a** and **b** are often forced to sum to 1 by writing one as some scalar *t* and the other as (1-*t*):

$$(1-t)\,\mathbf{a} + (t)\,\mathbf{b} \qquad (4.4)$$

Affine combinations of vectors appear in various contexts, as do affine combinations of points, as we see later.

Convex Combinations of Vectors.

Convex combinations have an important place in mathematics, and numerous applications in graphics. A **convex combination** arises as a further restriction on an affine combination. Not only must the coefficients of the linear combination sum to one; each one must also be nonnegative. The linear combination of Equation (4.2.2) is **convex** if:

$$a_1 + a_2 + \ldots + a_m = 1, \qquad (4.5)$$

and $a_i \geq 0$, for $i = 1,\ldots,m$.. As a consequence all a_i must lie between 0 and 1. (Why?).

Thus .3**a**+.7**b** is a convex combination of **a** and **b**, but 1.8**a** -.8**b** is not. The set of coefficients $a_1, a_2, \ldots, a_m$ is sometimes said to form a **partition of unity**, suggesting that a unit amount of "material" is partitioned into pieces. Convex combinations frequently arise in applications when one is making a unit amount of some brew and can combine only positive amounts of the various ingredients. They appear in unexpected contexts. For instance, we shall see in Chapter 8 that "spline" curves are in fact convex combinations of certain vectors, and in our discussion of color in Chapter 12 we shall find that colors can be considered as vectors, and that any color of unit brightness may be considered to be a convex combination of three primary colors!

We will find it useful to talk about the "set of all convex combinations" of a collection of vectors. Consider the set of all convex combinations of the two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$. It is the set of all vectors

$$\mathbf{v} = (1 - a)\,\mathbf{v}_1 + a\,\mathbf{v}_2 \qquad (4.6)$$

as the parameter *a* is allowed to vary from 0 to 1 (why?) What is this set? Rearranging the equation, **v** is seen to be:

$$\mathbf{v} = \mathbf{v}_1 + a\ (\mathbf{v}_2 - \mathbf{v}_1) \qquad (4.7)$$

Figure 4.8a shows this to be the vector that is $\mathbf{v}_1$ plus some fraction of $\mathbf{v}_2 - \mathbf{v}_1$, so the tip of $\mathbf{v}$ lies on the line joining $\mathbf{v}_1$ and $\mathbf{v}_2$. As a varies from 0 to 1, $\mathbf{v}$ takes on all the positions on the line from $\mathbf{v}_1$ to $\mathbf{v}_2$, and only those.

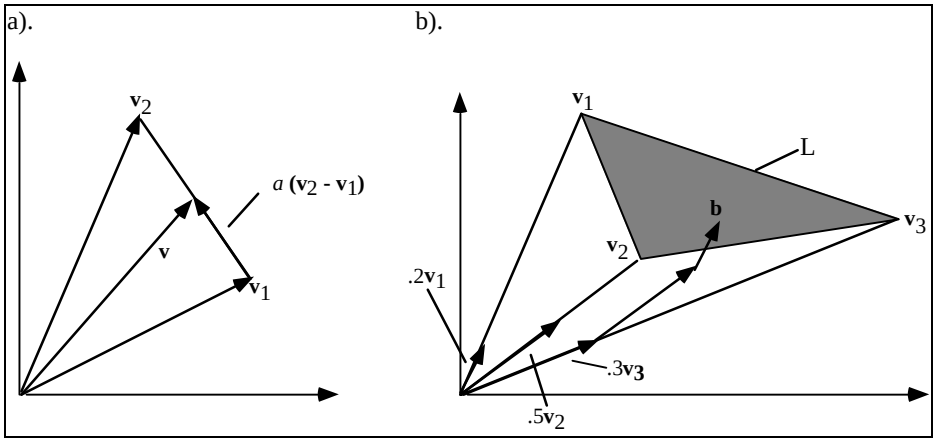


Figure 4.8. The set of vectors representable by convex combinations.

Figure 4.8b shows the set of all convex combinations of three vectors. Choose two parameters a_1 and a_2, both lying between 0 and 1, and form the following linear combination:

$$\mathbf{q} = a_1\,\mathbf{v}_1 + a_2\,\mathbf{v}_2 + (1 - a_1 - a_2)\,\mathbf{v}_3 \qquad (4.8)$$

where we also insist that a_1 plus a_2 does not exceed one. This is a convex combination, since none of the coefficients is ever negative and they sum to one. Figure 4.9 shows the three position vectors $\mathbf{v}_1 = (2, 6)$, $\mathbf{v}_2 = (3, 3)$, and $\mathbf{v}_3 = (7, 4)$. By the proper choices of a_1 and a_2, any vector lying within the shaded triangle of vectors can be represented, and no vectors outside this triangle can be reached. The vector $\mathbf{b} = .2\,\mathbf{v}_1 + .5\,\mathbf{v}_2 + .3\,\mathbf{v}_3$, for instance, is shown explicitly as the vector sum of the three weighted ingredients. (Note how it is built up out of "portions" of the three constituent vectors.) So the set of all convex combinations of these three vectors "spans" the shaded triangle. The proof of this is requested in the exercises.

If $a_2 = 0$, any vector in the line L that joins $\mathbf{v}_1$ and $\mathbf{v}_3$ can be "reached" by the proper choice of a_1. For example, the vector that is 20 percent of the way from $\mathbf{v}_1$ to $\mathbf{v}_3$ along L is given by $.8\,\mathbf{v}_1 + 0\,\mathbf{v}_2 + .2\,\mathbf{v}_3$.

4.2.3. The Magnitude of a vector, and unit vectors.

If a vector $\mathbf{w}$ is represented by the n-tuple $(w_1, w_2, \ldots, w_n)$, how might its magnitude (equivalently, its *length* or *size*) be defined and computed? We denote the magnitude by $|\mathbf{w}|$ and define it as the distance from its tail to its head. Based on the Pythagorean theorem, this becomes

$$|\mathbf{w}| = \sqrt{w_1^2 + w_2^2 + \ldots + w_n^2} \qquad (4.9)$$

For example, the magnitude of $\mathbf{w} = (4, -2)$ is $\sqrt{20}$, and that of $\mathbf{w} = (1, -3, 2)$ is $\sqrt{14}$. A vector of zero length is denoted as $\mathbf{0}$. Note that if $\mathbf{w}$ is the vector from point A to point B, then $|\mathbf{w}|$ will be the distance from A to B (why?).

It is often useful to scale a vector so that the result has a length equal to one. This is called **normalizing** a vector, and the result is known as a **unit vector**. For example, we form the normalized version of $\mathbf{a}$, denoted $\hat{\mathbf{a}}$, by scaling it with the value $1/|\mathbf{a}|$:

$$\hat{\mathbf{a}} = \frac{\mathbf{a}}{|\mathbf{a}|} \qquad (4.10)$$

Clearly this is a unit vector: $|\hat{\mathbf{a}}| = 1$ (why?), having the same direction as **a**. For example, if **a** = (3, -4), then | **a** | = 5 and the normalized version is $\hat{\mathbf{a}} = (\frac{3}{5}, \frac{-4}{5})$. At times we refer to a unit vector as a **direction**. Note that any vector can be written as its magnitude times its direction: If $\hat{\mathbf{a}}$ is the normalized version of **a**, vector **a** may always be written $\mathbf{a} = |\mathbf{a}|\,\hat{\mathbf{a}}$.

Practice Exercises.

4.2.1. Representing Vectors as linear combinations. With reference to Figure 4.9, what values, or range of values, for a_1 and a_2 create the following sets?

a. $\mathbf{v}_1$.

b. The line joining $\mathbf{v}_1$ and $\mathbf{v}_2$.

c. The vector midway between $\mathbf{v}_2$ and $\mathbf{v}_3$.

d. The centroid of the triangle.

4.2.2. The set of all convex combinations. Show that the set of all convex combinations of three vectors $\mathbf{v}_1$, $\mathbf{v}_2$, and $\mathbf{v}_3$ is the set of vectors whose tips lie in the "triangle" formed by the tips of the three vectors. Hint: Each point in the triangle is a combination of $\mathbf{v}_1$ and some point lying between $\mathbf{v}_2$ and $\mathbf{v}_3$.

4.2.3. Factoring out a scalar. Show how scaling a vector **v** by a scalar *s* changes its length. That is, show that: | s **v** | = | s | | **v** | . Note the dual use of the magnitude symbol | |, once for a scalar and once for a vector.

4.2.4. Normalizing Vectors. Normalize each of the following vectors:

a). (1, -2, .5); b). (8, 6); c). (4, 3)

4.3. The Dot Product.

There are two other powerful tools that facilitate working with vectors: the dot (or inner) product, and the cross product. The dot product produces a scalar; the cross product works only on three dimensional vectors and produces another vector. In this section we review the basic properties of the dot product, principally to develop the notion of perpendicularity. We then work with the dot product to solve a number of important geometric problems in graphics. Then the cross product is introduced, and used to solve a number of 3D geometric problems.

The **dot product** of two vectors is simple to define and compute. For two-dimensional vectors, (a_1, a_2) and (b_1, b_2), it is simply the scalar whose value is $a_1b_1 + a_2b_2$. Thus to calculate it, multiply corresponding components of the two vectors, and add the results. For example, the dot product of (3, 4) and (1, 6) is 27, and that of (2, 3) and (9, -6) is 0.

The definition of the dot product generalizes easily to *n* dimensions:

Definition: The Dot Product

The dot product *d* of two *n*-dimensional vectors $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ and $\mathbf{w} = (w_1, w_2, \ldots, w_n)$ is denoted as $\mathbf{v} \cdot \mathbf{w}$ and has the value

$$d = \mathbf{v} \cdot \mathbf{w} = \sum_{i=1}^{n} v_i w_i \qquad (4.11)$$

Example 4.3.1:

- The dot product of (2, 3, 1) and (0, 4, -1) is 11.
- (2, 2, 2, 2) · (4, 1, 2, 1.1) = 16.2.
- (1, 0, 1, 0, 1) · (0, 1, 0, 1, 0) = 0.
- (169, 0, 43) · (0, 375.3, 0) = 0.

4.3.1. Properties of the Dot Product

The dot product exhibits four major properties that we frequently exploit and that follow easily (see the exercises) from its basic definition:

1·	Symmetry:	$\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$
2·	Linearity:	$(\mathbf{a} + \mathbf{c}) \cdot \mathbf{b} = \mathbf{a} \cdot \mathbf{b} + \mathbf{c} \cdot \mathbf{b}$
3·	Homogeneity:	$(s\mathbf{a}) \cdot \mathbf{b} = s\,(\mathbf{a} \cdot \mathbf{b})$
4·	$\lvert\mathbf{b}\rvert^2 = \mathbf{b} \cdot \mathbf{b}$	

The first states that the order in which the two vectors are combined does not matter: the dot product is **commutative**. The next two proclaim that the dot product is **linear**; that is, the dot product of a sum of vectors can be expressed as the sum of the individual dot products, and scaling a vector scales the value of the dot product. The last property is also useful, as it asserts that taking the dot product of a vector with itself yields the **square of the length** of the vector. It appears frequently in the form $|\mathbf{b}| = \sqrt{\mathbf{b} \cdot \mathbf{b}}$.

The following manipulations show how these properties can be used to simplify an expression involving dot products. The result itself will be used in the next section.

Example 4.3.2: Simplification of $|\mathbf{a} - \mathbf{b}|^2$.
Simplify the expression for the length (squared) of the difference of two vectors, **a** and **b**, to obtain the following relation:

$$| \mathbf{a} - \mathbf{b} |^2 = | \mathbf{a} |^2 - 2\,\mathbf{a} \cdot \mathbf{b} + | \mathbf{b} |^2 \qquad (4.12)$$

The derivation proceeds as follows: Give the name *C* to the expression $| \mathbf{a} - \mathbf{b} |^2$. By the fourth property, *C* is the dot product:

$C = | \mathbf{a} - \mathbf{b} |^2 = (\mathbf{a} - \mathbf{b}) \cdot (\mathbf{a} - \mathbf{b})$.
Using linearity: $C = \mathbf{a} \cdot (\mathbf{a} - \mathbf{b}) - \mathbf{b} \cdot (\mathbf{a} - \mathbf{b})$.
Using symmetry and linearity to simplify this further: $C = \mathbf{a} \cdot \mathbf{a} - 2\mathbf{a} \cdot \mathbf{b} + \mathbf{b} \cdot \mathbf{b}$.
Using the fourth property above to obtain $C = | \mathbf{a} |^2 - 2\,\mathbf{a} \cdot \mathbf{b} + | \mathbf{b} |^2$ gives the desired result.

By replacing the minus with a plus in this relation, the following similar and useful relation emerges:

$$| \mathbf{a} + \mathbf{b} |^2 = | \mathbf{a} |^2 + 2\mathbf{a} \cdot \mathbf{b} + | \mathbf{b} | \qquad (4.13)$$

4.3.2. The Angle Between Two Vectors.

The most important application of the dot product is in finding the angle between two vectors, or between two intersecting lines. Figure 4.9 shows the 2D case, where vectors **b** and **c** lie at angles $\varphi_{\mathbf{b}}$, and $\varphi_{\mathbf{c}}$, relative to the *x*-axis. Now from elementary trigonometry:

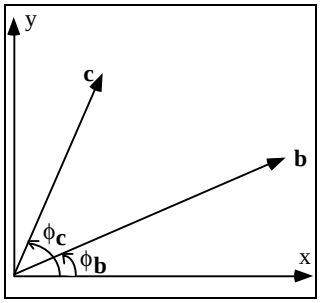


Figure 4.9. Finding the angle between two vectors.

$\mathbf{b} = (|\mathbf{b}| \cos \varphi_{\mathbf{b}} , |\mathbf{b}| \sin \varphi_{\mathbf{b}})$
$\mathbf{c} = (|\mathbf{c}| \cos \varphi_{\mathbf{c}} , |\mathbf{c}| \sin \varphi_{\mathbf{c}})$.

Thus their dot product is

$$\mathbf{b} \cdot \mathbf{c} = |\mathbf{b}|\,|\mathbf{c}| \cos\varphi_{\mathbf{c}} \cos\varphi_{\mathbf{b}} + |\mathbf{b}|\,|\mathbf{c}| \sin\varphi_{\mathbf{b}} \sin\varphi_{\mathbf{c}}$$
$$= |\mathbf{b}||\mathbf{c}| \cos(\varphi_{\mathbf{c}} - \varphi_{\mathbf{b}})$$

so we have, for any two vectors **b** and **c**:

$\mathbf{b} \cdot \mathbf{c} = |\mathbf{b}|\,|\mathbf{c}| \cos(\theta)$ (4.14)

where θ is the angle from **b** to **c**. Thus $\mathbf{b} \cdot \mathbf{c}$ varies as the cosine of the angle from **b** to **c**. The same result holds for vectors of three, four, or any number of dimensions.

To obtain a slightly more compact form, divide through both sides by $|\mathbf{b}|\,|\mathbf{c}|$ and use the unit vector notation $\hat{\mathbf{b}} = \mathbf{b} / |\mathbf{b}|$ to obtain

$$\cos(\theta) = \hat{\mathbf{b}} \cdot \hat{\mathbf{c}} \qquad (4.15)$$

This is the desired result: The cosine of the angle between two vectors **b** and **c** is the dot product of their normalized versions.

Example 4.3.3. Find the angle between **b** = (3, 4) and **c** = (5, 2).
Solution: Form |**b**| = 5 and |**c**| = 5.385 so that $\hat{\mathbf{b}}$ = (3 / 5, 4 / 5) and $\hat{\mathbf{c}}$ = (.9285, .3714). The dot product $\hat{\mathbf{b}} \cdot \hat{\mathbf{c}}$ = .85422 = cos(θ), so that θ = 31.326°. This can be checked by plotting the two vectors on graph paper and measuring the angle between them.

4.3.3. The Sign of b·c, and Perpendicularity.

Recall that cos(θ) is **positive** if |θ| is less than 90°, **zero** if |θ| equals 90°, and **negative** if |θ| exceeds 90°. Because the dot product of two vectors is proportional to the cosine of the angle between them, we can therefore observe immediately that two vectors (of any nonzero length) are

- less than 90° apart if $\mathbf{b} \cdot \mathbf{c} > 0$;
- exactly 90° apart if $\mathbf{b} \cdot \mathbf{c} = 0$; (4.16)
- more than 90° apart if $\mathbf{b} \cdot \mathbf{c} < 0$;

This is indicated by Figure 4.10. The sign of the dot product is used in many algorithmic tests.

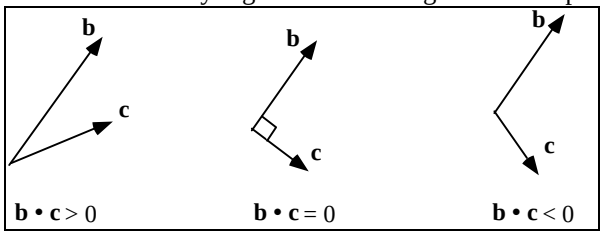


Figure 4.10. The sign of the dot product.

The case in which the vectors are 90° apart, or **perpendicular**, is of special importance.

Definition:
Vectors **b** and **c** are perpendicular if $\mathbf{b} \cdot \mathbf{c} = 0$.
(4.17)

Other names for "perpendicular" are **orthogonal** and **normal**, and we shall use all three interchangeably.

The most familiar examples of orthogonal vectors are those aimed along the axes of 2D and 3D coordinate systems, as shown in Figure 4.11. In part a) the 2D vectors (1, 0) and (0, 1) are mutually perpendicular unit vectors. The 3D versions are so commonly used they are called the "standard unit vectors" and are given names **i**, **j**, and **k**.

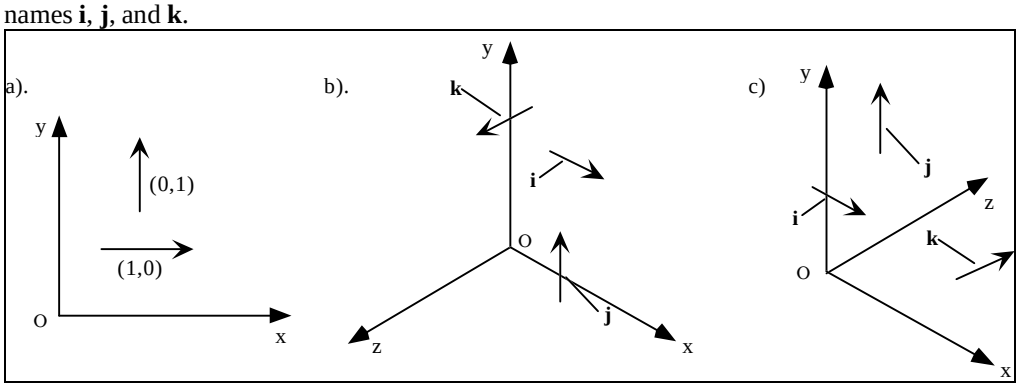


Figure 4.11. The standard unit vectors.

Definition:
The **standard unit vectors** in 3D have components:
i = (1, 0, 0), **j** = (0, 1, 0), and **k** = (0, 0, 1). (4.18)

Part b) of the figure shows them for a right-handed system, and part c) shows them for a left-handed system. Note that **k** always points in the positive z direction.

Using these definitions any 3D vector such as (*a*, *b*, *c*) can be written in the alternative form:

$$(a, b, c) = a\,\mathbf{i} + b\,\mathbf{j} + c\,\mathbf{k} \qquad (4.19)$$

Example 4.3.4. Notice that **v** = (2, 5, -1) is clearly the same as 2 (1, 0, 0) + 5 (0, 1, 0) -1 (0, 0, 1), which is recognized as 2 **i** + 5 **j** - **k**.

This form presents a vector as a sum of separate elementary component vectors, so it simplifies various pencil-and-paper calculations. It is particularly convenient when dealing with the cross product, discussed in Section 4.4.

Practice Exercises.
4.3.1. Alternate proof of $\mathbf{b} \cdot \mathbf{c} = |\mathbf{b}|\,|\mathbf{c}| \cos\theta$. Note that **b** and **c** form two sides of a triangle, and the third side is **b** - **c**. Use the law of cosines to obtain the square of the length of **b** - **c** in terms of the lengths of **b** and **c** and the cosine of θ. Compare this with Equation 4.13 to obtain the desired result.
4.3.2. Find the Angle. Calculate the angle between the vectors (2, 3) and (-3, 1), and check the result visually using graph paper. Then compute the angle between the 3D vectors (1, 3, -2) and (3, 3, 1).
4.3.3. Testing for Perpendicularity. Which pairs of the following vectors are perpendicular to one another: (3, 4, 1), (2, 1, 1), (-3, -4, 1), (0, 0, 0), (1, -2, 0), (4, 4, 4), (0, -1, 4), and (2, 2, 1)?
4.3.4. Pythagorean Theorem. Refer to Equations 4.12 and 4.13. For the case in which **a** and **b** are perpendicular, these expressions have the same value, which seems to make no sense geometrically. Show that it works all right, and relate the result to the Pythagorean theorem.

4.3.4. The 2D "Perp" Vector.

Suppose the 2D vector **a** has components (a_x, a_y). What vectors are perpendicular to it? One way to obtain such a vector is to interchange the *x*- and *y*- components and negate one of them.[3] Let **b** = ($-a_y$, a_x). Then the dot product **a** · **b** equals 0 so **a** and **b** are indeed perpendicular. For instance, if **a** = (4,7) then **b** = (-7, 4) is a vector normal to **a**. There are infinitely many vectors normal to any **a**, since any scalar multiple of **b**, such as (-21, 12) and (7, -4) is also normal to **a**. (Sketch several of them for a given **a**.)

It is convenient to have a symbol for one *particular* vector that is normal to a given 2D vector **a**. We use the symbol $\perp$ (pronounced "perp") for this.

Definition: Given $\mathbf{a} = (a_x, a_y)$, $\mathbf{a}^\perp = (-a_y, a_x)$ is the **counterclockwise** **perpendicular** to **a**. (4.20)

Note that **a** and $\mathbf{a}^\perp$ have the same length: $|\mathbf{a}| = |\mathbf{a}^\perp|$. Figure 4.12a shows an arbitrary vector **a** and the resulting $\mathbf{a}^\perp$. Note that moving from the **a** direction to direction $\mathbf{a}^\perp$ requires a left turn. (Making a right turn is equivalent to turning in the direction $-\mathbf{a}^\perp$.)

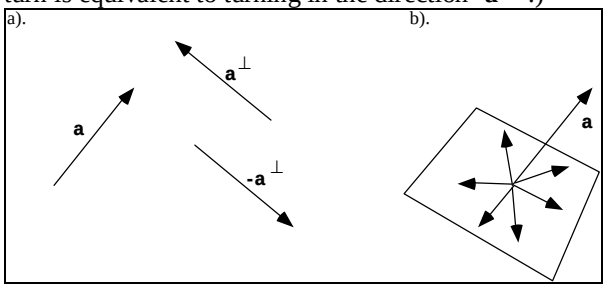


Figure 4.12. The vector $\mathbf{a}^\perp$ perpendicular to **a**.

We show in the next section how this notation can be put to good use. Figure 4.12b shows that in three dimensions no single vector lies in "the" direction perpendicular to a given 3D vector **a**, since any of the vectors lying in the plane perpendicular to **a** will do. However, the cross product developed later will provide a simple tool for dealing with such vectors.

Practice Exercises.

4.3.5. Some Pleasant Properties of $\mathbf{a}^\perp$. It is useful in some discussions to view the "perp" symbol $\perp$ as an operator that performs a "rotate 90° left" operation on its argument, so that $\mathbf{a}^\perp$ is the vector produced by applying the $\perp$ to vector **a**, much as $\sqrt{x}$ is the value produced by applying the square root operator to *x*. Viewing $\perp$ in this way, show that it enjoys the following properties:

a). Linearity: $(\mathbf{a} + \mathbf{b})^\perp = \mathbf{a}^\perp + \mathbf{b}^\perp$ and $(A\mathbf{a})^\perp = A\mathbf{a}^\perp$ for any scalar *A*;

b). $\mathbf{a}^{\perp\perp} = (\mathbf{a}^\perp)^\perp = -\mathbf{a}$ (two perp's make a reversal)

4.3.6. The "perp dot" product. Interesting things happen when we dot the "perp" of a vector with another vector, as in $\mathbf{a}^\perp \cdot \mathbf{b}$. We call this the "perp dot product" [hill95]. Use the basic definition of $\mathbf{a}^\perp$ above to show:

- $\mathbf{a}^\perp \cdot \mathbf{b} = a_x b_y - a_y b_x$ (value of the perp dot product)
- $\mathbf{a}^\perp \cdot \mathbf{a} = 0$, ($\mathbf{a}^\perp$ is perpendicular to **a**)
- $|\mathbf{a}^\perp|^2 = |\mathbf{a}|^2$. ($\mathbf{a}^\perp$ and **a** have the same length) (4.21)
- $\mathbf{a}^\perp \cdot \mathbf{b} = -\mathbf{b}^\perp \cdot \mathbf{a}$, (antisymmetric)

[3] This is equivalent to the familiar fact that perpendicular lines have slopes that are negative reciprocals of one another. In Chapter 5 we see the "interchange and negate" operation arise naturally in connection with a rotation of 90 degrees.

The fourth fact shows that the perp dot product is "antisymmetric": moving the $^\perp$ from one vector to the other reverses the sign of the dot product. Other useful properties of the perp dot product will be discussed as they are needed.

4.3.7. Calculate one. Compute $\mathbf{a} \cdot \mathbf{b}$ and $\mathbf{a}^\perp \cdot \mathbf{b}$ for $\mathbf{a} = (3,4)$ and $\mathbf{b} = (2,1)$.

4.3.8. It's a determinant. Show that $\mathbf{a}^\perp \cdot \mathbf{b}$ can be written as the determinant (for definitions of matrices and determinants see Appendix 2):

$$\mathbf{a}^\perp \cdot \mathbf{b} = \begin{vmatrix} a_x & a_y \\ b_x & b_y \end{vmatrix}$$

4.3.9. Other goodies.

a). Show that $(\mathbf{a}^\perp \cdot \mathbf{b})^2 + (\mathbf{a} \cdot \mathbf{b})^2 = |\mathbf{a}|^2 |\mathbf{b}|^2$.

b). Show that if $\mathbf{a} + \mathbf{b} + \mathbf{c} = \mathbf{0}$ then $\mathbf{a}^\perp \cdot \mathbf{b} = \mathbf{b}^\perp \cdot \mathbf{c} = \mathbf{c}^\perp \cdot \mathbf{a}$.

4.3.5. Orthogonal Projections, and the Distance from a Point to a Line.

Three geometric problems arise frequently in graphics applications: **projecting** a vector onto a given vector, **resolving** a vector into its components in one direction and another, and finding the distance between a point and a line. All three problems are simplified if we use the perp vector and the perp dot product.

Figure 4.13a shows the basic ingredients. We are given two points *A* and *C*, and a vector **v**. These questions arise:

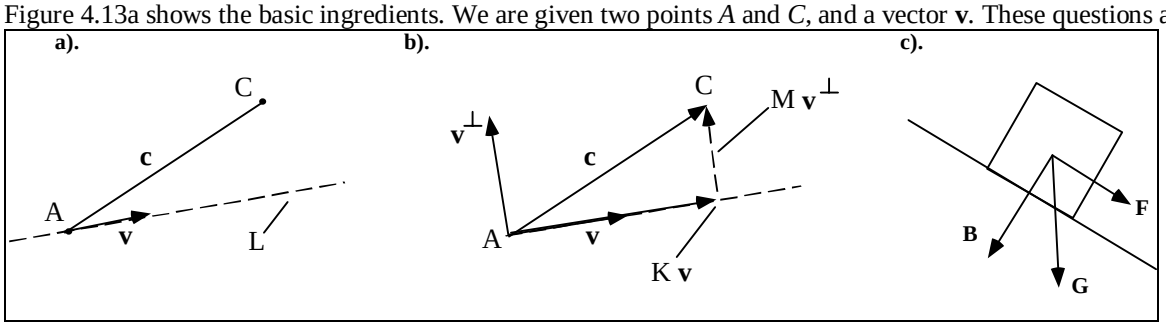


Figure 4.13. Resolving a vector into two orthogonal vectors.

a). How far is the point *C* from the line *L* that passes through *A* in the direction **v**?
b). If we drop a perpendicular from *C* onto *L*, where does it hit *L*?
c). How do we decompose the vector $\mathbf{c} = C - A$ into a part along the line *L* and a part perpendicular to *L*?

Figure 4.13.b defines some additional quantities: $\mathbf{v}^\perp$ is the vector **v** rotated 90 degrees CCW. Dropping a perpendicular from *C* onto line *L* we say that the vector **c** is **resolved** into the portion $K\mathbf{v}$ along **v** and the portion $M\,\mathbf{v}^\perp$ perpendicular to **v**, where *K* and *M* are some constants to be determined. Then we have

$$\mathbf{c} = K\mathbf{v} + M\,\mathbf{v}^\perp \qquad (4.22)$$

Given **c** and **v** we want to solve for *K* and *M*. Once found, we say that the **orthogonal projection** of **c** onto **v** is K**v**, and that the distance from C to the line is $|M\mathbf{v}^\perp|$.

Figure 4.13c shows a situation where these questions might arise. We wish to analyze how the gravitational force vector **G** acts on the block to pull it down the incline. To do this we must resolve **G** into the force **F** acting along the incline and the force **B** acting perpendicular to the incline. That is, find **F** and **B** such that $\mathbf{G} = \mathbf{F} + \mathbf{B}$.

Equation 4.22 is really two equations: the left and right hand sides must agree for the *x*-components and they also must agree for the *y*-components. There are two unknowns *K* and *M*. So we have two equations in two unknowns, and Cramer's rule can be applied. But who remembers Cramer's rule? We use a trick

here that is easy to remember and immediately reveals the solution. It is equivalent to Cramer's rule, but simpler to apply.

The trick in solving two equations in two unknowns is to eliminate one of the variables. We do this by forming the dot product of both sides with the vector **v**:

$$\mathbf{c}\cdot\mathbf{v} = K\mathbf{v}\cdot\mathbf{v} + M\,\mathbf{v}^{\perp}\cdot\mathbf{v} \qquad (4.23)$$

Happily, the term $\mathbf{v}^{\perp}\cdot\mathbf{v}$ vanishes, (why?), yielding *K* immediately:

$$K = \frac{\mathbf{c}\cdot\mathbf{v}}{\mathbf{v}\cdot\mathbf{v}}.$$

Similarly "dot" both sides of Equation 4.3.12 with $\mathbf{v}^{\perp}$ to obtain M:

$$M = \frac{\mathbf{c}\cdot\mathbf{v}^{\perp}}{\mathbf{v}\cdot\mathbf{v}}$$

where we have used the third property in Equation 4.21. Putting these together we have

$$\mathbf{c} = \left(\frac{\mathbf{v}\cdot\mathbf{c}}{|\mathbf{v}|^2}\right)\mathbf{v} + \left(\frac{\mathbf{v}^{\perp}\cdot\mathbf{c}}{|\mathbf{v}|^2}\right)\mathbf{v}^{\perp} \qquad \text{(resolving } \mathbf{c} \text{ into } \mathbf{v} \text{ and } \mathbf{v}^{\perp}\text{)} \qquad (4.24)$$

This equality holds for any vectors **c** and **v.** The part along **v** is known as the **orthogonal projection** of **c** onto the vector **v**. The second term gives the "difference term" explicitly and compactly. Its size is the distance from C to the line:

$$\text{distance} = \left|\frac{\mathbf{v}^{\perp}\cdot\mathbf{c}}{|\mathbf{v}|^2}\,\mathbf{v}^{\perp}\right| = \frac{|\mathbf{v}^{\perp}\cdot\mathbf{c}|}{|\mathbf{v}|},$$

(Check that the second form really equals the first). Referring to Figure 4.13b we can say: **the distance from a point** C **to the line through** *A* in the direction **v** is:

$$\text{distance} = \frac{|\mathbf{v}^{\perp}\cdot(C-A)|}{|\mathbf{v}|}. \qquad (4.25)$$

Example 4.3.5. Find the orthogonal projection of the vector **c** = (6, 4) onto **a** = (1, 2). (Sketch the relevant vectors.) **Solution**: Evaluate the first term in Equation 4.24, obtaining the vector (14, 28) / 5.
Example 4.3.6: How far is the point *C* = (6,4) from the line that passes through (1,1) and (4,9)? **Solution**: Set *A* = (1, 1), use **v** = (4, 9) - (1, 1) = (3, 8), and evaluate *distance* in Equation 4.25. The result is: $d = 31/\sqrt{73}$.

Practice Exercises.

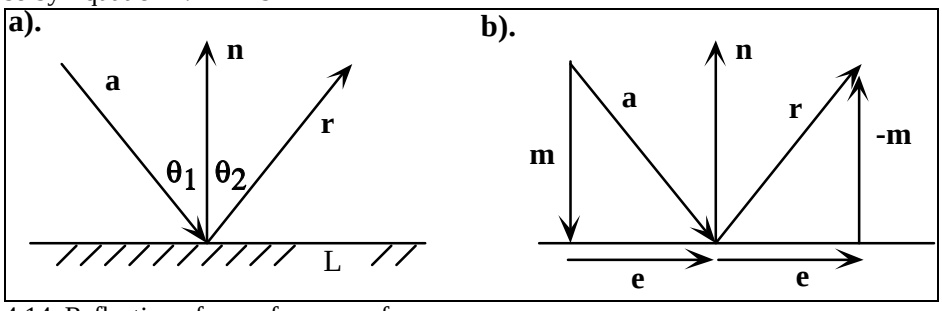
4.3.10. Resolve it. Express vector **g** = (4, 7) as a linear combination of **b** = (3, 5) and $\mathbf{b}^{\perp}$. How far is (4, 2) + **g** from the line through (4, 2) that moves in the direction **b**?
4.3.11. A Block pulled down an incline. A block rests on an incline tilted 30° from the horizontal. Gravity exerts a force of one newton on the block. What is the force that is "trying" to move the block along the incline?
4.3.12. How far is it? How far from the line through (2, 5) and (4, -1) does the point (6, 11) lie? Check your result on graph paper.

4.3.6. Applications of Projection: Reflections.

To display the reflection of light from a mirror, or the behavior of billiard balls bouncing off one another, we need to find the direction that an object takes upon being reflected at a given surface. In a case study at the end of this chapter we describe an application to trace a ray of light as it bounces around inside a reflective chamber, or a billiard ball as it bounces around a pool table. At each bounce a reflection is made to a new direction, as derived in this section.

When light reflects from a mirror we know that the angle of reflection must equal the angle of incidence. We next show how to use vectors and projections to compute this new direction. We can think in terms of two-dimensional vectors for simplicity, but because the derivation does not explicitly state the dimension of the vectors involved, the same result applies in three dimensions for reflections from a surface.

Figure 4.14a shows a ray having direction **a**, hitting line *L*, and reflecting in (as yet unknown) direction **r**. The vector **n** is perpendicular to the line. Angle θ_1 in the figure must equal angle θ_2. How is **r** related to **a** and **n**? Figure 4.14b shows **a** resolved into the portion **m** along **n** and the portion **e** orthogonal to **n**. Because of symmetry, **r** has the same component **e** orthogonal to **n**, but the opposite component along **n**, and so **r** = **e** - **m**. Because **e** = **a** - **m**, this gives **r** = **a** - 2**m**. Now **m** is the orthogonal projection of **a** onto **n**, so by Equation 4.24 **m** is



4.14. Reflection of a ray from a surface.

$$\mathbf{m} = \frac{\mathbf{a}\cdot\mathbf{n}}{|\mathbf{n}|^2}\mathbf{n} = (\mathbf{a}\cdot\hat{\mathbf{n}})\hat{\mathbf{n}} \qquad (4.26)$$

(recall $\hat{\mathbf{n}}$ is the unit length version of **n**) and so we obtain the result

$$\mathbf{r} = \mathbf{a} - 2(\mathbf{a}\cdot\hat{\mathbf{n}})\hat{\mathbf{n}} \qquad \text{(direction of the reflected ray)} \qquad (4.27)$$

In three dimensions physics demands that the reflected direction **r** must lie in the plane defined by **n** and **a**. The expression for **r** above indeed supports this, as we show in Chapter five.

Example 4.3.7. Let **a** = (4, -2) and **n** = (0, 3). Then Equation 4.27 yields **r** = (4, 2), as expected. Both the angle of incidence and reflection are equal to $\tan^{-1}(2)$.

Practice Exercises.
4.3.13. Find the Reflected Direction. For **a** = (2, 3) and **n** = (-2, 1), find the direction of the reflection.
4.3.14. Lengths of the Incident and Reflected Vectors. Using Equation 4.27 and properties of the dot product, show that |**r**| = |**a**|.

4.4. The Cross Product of Two Vectors.

The **cross product** (also called the **vector product**) of two vectors is another vector. It has many useful properties, but the one we use most often is that it is perpendicular to both of the given vectors. The cross product is defined only for three-dimensional vectors.

Given the 3D vectors **a** = (a_X, a_Y, a_Z) and **b** = (b_X, b_Y, b_Z), their cross product is denoted as $\mathbf{a} \times \mathbf{b}$. It is defined in terms of the standard unit vectors **i**, **j**, and **k** (see Equation 4.18) by

Definition of $\mathbf{a} \times \mathbf{b}$:

$$\mathbf{a} \times \mathbf{b} = (a_y b_z - a_z b_y)\mathbf{i} + (a_z b_x - a_x b_z)\mathbf{j} + (a_x b_y - a_y b_x)\mathbf{k} \qquad (4.28)$$

(It can actually be derived from more fundamental principles: See the exercises.) As this form is rather difficult to remember, it is often written as an easily remembered determinant (see Appendix 2 for a review of determinants).

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} \qquad (4.29)$$

Remembering how to form the cross product thus requires only remembering how to form a determinant.

Example 4.4.1. For **a** = (3, 0, 2) and **b** = (4, 1, 8), direct calculation shows that **a**× **b** = -2**i** -16**j** + 3**k**. What is **b**× **a**?

From this definition one can easily show the following algebraic properties of the cross product:

1. $\mathbf{i} \times \mathbf{j} = \mathbf{k}$, $\mathbf{j} \times \mathbf{k} = \mathbf{i}$, $\mathbf{k} \times \mathbf{i} = \mathbf{j}$

2. $\mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a}$ (antisymmetry)
3. $\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c}$ (linearity) (4.30)
4. $(s\mathbf{a}) \times \mathbf{b} = s(\mathbf{a} \times \mathbf{b})$ (homogeneity)

These equations are true in both left-handed and right-handed coordinate systems. Note the logical (alphabetical) ordering of ingredients in the equation $\mathbf{i} \times \mathbf{j} = \mathbf{k}$, which also provides a handy mnemonic device for remembering the direction of cross products.

Practice Exercises.

4.4.1. Demonstrate the Four Properties. Prove each of the preceding four properties given for the cross product.

4.4.2. Derivation of the Cross Product. The form in Equation 4.28, presented as a definition, can actually be derived from more fundamental ideas. We need only assume that:

a. The cross product operation is linear.

b. The cross product of a vector with itself is 0.

c. $\mathbf{i} \times \mathbf{j} = \mathbf{k}$, $\mathbf{j} \times \mathbf{k} = \mathbf{i}$, and $\mathbf{k} \times \mathbf{i} = \mathbf{j}$.

By writing **a** = a_X **i** + a_Y **j** + a_Z **k** and **b** = b_X **i** + b_Y **j** + b_Z **k**, apply these rules to derive the proper form for $\mathbf{a} \times \mathbf{b}$.

4.4.3. Is $\mathbf{a} \times \mathbf{b}$ perpendicular to a? Show that the cross product of vectors **a** and **b** is indeed perpendicular to **a**.

4.4.4. Vector Products. Find the vector **b** = (b_X, b_Y, b_Z) that satisfies the cross product relation $\mathbf{a} \times \mathbf{b}$ = **c**, where **a** = (2, 1, 3) and **c** = (2, -4, 0). Is there only one such vector?

4.4.5. Nonassociativity of the Cross Product. Show that the cross product is not associative. That is, that $\mathbf{a} \times (\mathbf{b} \times \mathbf{c})$ is not necessarily the same as $(\mathbf{a} \times \mathbf{b}) \times \mathbf{c}$.

4.4.6. Another Useful Fact. Show by direct calculation on the components that the length of the cross product has the form:

$$|\mathbf{a} \times \mathbf{b}| = \sqrt{|\mathbf{a}|^2 |\mathbf{b}|^2 - (\mathbf{a} \cdot \mathbf{b})^2}$$

4.4.1. Geometric Interpretation of the Cross Product.

By definition the cross product $\mathbf{a} \times \mathbf{b}$ of two vectors is another vector, but how is it related geometrically to the others, and why is it of interest? Figure 4.15 gives the answer. The cross product $\mathbf{a} \times \mathbf{b}$ has the following useful properties (whose proofs are requested in the exercises):

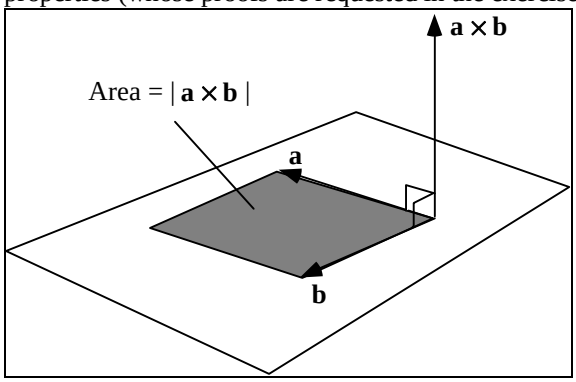


Figure 4.15. Interpretation of the cross product.

1. $\mathbf{a} \times \mathbf{b}$ is perpendicular (orthogonal) to both **a** and **b**.

2. The length of $\mathbf{a} \times \mathbf{b}$ equals the area of the parallelogram determined by **a** and **b**. This area is equal to

$$|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}||\mathbf{b}|\sin(\theta) \qquad (4.31)$$

where θ is the angle between **a** and **b**, measured from **a** to **b** or **b** to **a**, whichever produces an angle less than 180 degrees. As a special case, $\mathbf{a} \times \mathbf{b} = 0$ if, and only if, **a** and **b** have the same or opposite directions or if either has zero length. What is the magnitude of the cross product if **a** and **b** are perpendicular?

3. The sense of $\mathbf{a} \times \mathbf{b}$ is given by the right-hand rule when working in a right-handed system. For example, twist the fingers of your right hand from **a** to **b**, and then $\mathbf{a} \times \mathbf{b}$ will point in the direction of your thumb. (When working in a left-handed system, use your left hand instead.) Note that $\mathbf{i} \times \mathbf{j} = \mathbf{k}$ supports this.

Example 4.4.2. Let **a** = (1, 0, 1) and **b** = (1, 0, 0). These vectors are easy to visualize, as they both lie in the *x*, *z*-plane. (Sketch them.) The area of the parallelogram defined by **a** and **b** is easily seen to be 1. Because $\mathbf{a} \times \mathbf{b}$ is orthogonal to both **a** and **b**, we expect it to be parallel to the *y*-axis and hence be proportional to $\pm\mathbf{j}$. In either a right-handed or a left-handed system, sweeping the fingers of the appropriate hand from **a** to **b** reveals a thumb pointed along the positive *y*-axis. Direct calculation based on Equation 4.28 confirms all of this: $\mathbf{a} \times \mathbf{b} = \mathbf{j}$.

Practice Exercise 4.4.7. Proving the Properties. Prove the three properties given above for the cross product.

4.4.2. Finding the Normal to a Plane.

As we shall see in the next section, we sometimes must compute the components of the normal vector **n** to a plane.

If the plane is known to pass through three specific points, the cross product provides the tool to accomplish this.

Any three points, P_1, P_2, P_3, determine a unique plane, as long as the points don't lie in a straight line. Figure 4.16 shows this situation.

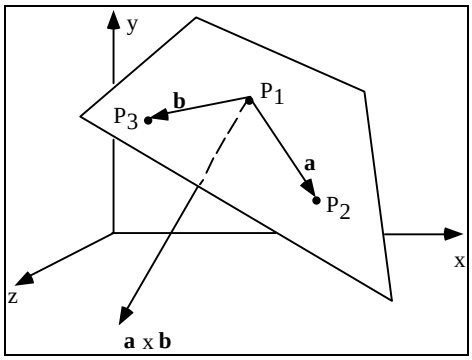


Figure 4.16. Finding the plane through three given points.

To find the normal vector, build two vectors, $\mathbf{a} = P_2 - P_1$ and $\mathbf{b} = P_3 - P_1$. Their cross product, $\mathbf{n} = \mathbf{a} \times \mathbf{b}$, must be normal to both **a** and **b**, so it is normal to every line in the plane (why?). It is therefore the desired normal vector. (What happens if the three points do lie in a straight line?) Any scalar multiple of this cross product is also a normal vector, including $\mathbf{b} \times \mathbf{a}$, which points in the opposite direction.

Example 4.4.3. Find the normal vector to the plane that passes through the points (1, 0, 2), (2, 3, 0), and (1, 2, 4).
Solution: By direct calculation, **a** = (2, 3, 0) - (1, 0, 2) = (1, 3, -2), and **b** = (1, 2, 4) - (1, 0, 2) = (0, 2, 2), and so their cross product **n** = (10, -2, 2).

Note: Since a cross product involves the subtraction of various quantities (see Equation 4.28), this method for finding **n** is vulnerable to numerical inaccuracies, especially when the angle between **a** and **b** is small. We develop a more robust method later for finding normal vectors in practice.

Practice Exercises.
4.4.8. Does the choice of points matter? Is the same plane obtained as in Example 4.4.3 if we use the points in a different order, say, **a** = (1, 0, 2) - (2, 3, 0) and **b** = (1, 2, 4) - (2, 3, 0)? Show that the same plane does result.
4.4.9. Finding Some Planes. For each of the following triplets of points, find the normal vector to the plane (if it exists) that passes through the triplet.
a. P_1 = (1, 1, 1), P_2 = (1, 2, 1), P_3 = (3, 0, 4)
b. P_1 = (8, 9, 7), P_2 = (-8, -9, -7), P_3 = (1, 2, 1)
c. P_1 = (6, 3, -4), P_2 = (0, 0, 0), P_3 = (2, 1, -1)
d. P_1 = (0, 0, 0), P_2 = (1, 1, 1), P_3 = (2, 2, 2).
4.4.10. Finding the normal vectors. Calculate the normal vectors to each of the faces of the two objects shown in Figure 4.17. The cube has vertices (±1,±1,±1) and the tetrahedron has vertices (0,0,0), (0,0,1), (1,0,0), and (0,1,0).

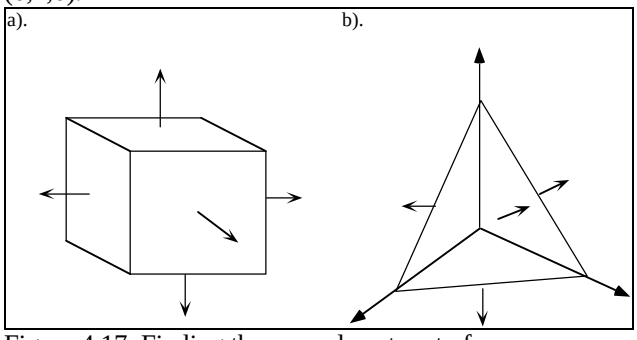


Figure 4.17. Finding the normal vectors to faces.

4.5. Representations of Key Geometric Objects.

In the preceding sections we have discussed some basic ideas of vectors and their application to important geometric problems that arise in graphics. Now we develop the fundamental ideas that facilitate working

with lines and planes, which are central to graphics, and whose "straightness" and "flatness" makes them easy to represent and manipulate.

What does it mean to "represent" a line or plane, and why is it important? The goal is to come up with a formula or equation that distinguishes points that lie on the line from those that don't. This might be an equation that is satisfied by all points on the line, and only those points. Or it might be a function that returns different points in the line as some parameter is varied. The representation allows one to test such things as: is point *P* on the line?, or where does the line *intersect* another line or some other object. Very importantly, a line lying in a plane divides the plane into two parts, and we often need to ask whether point *P* lies on one side or the other of the line.

In order to deal properly with lines and planes we must, somewhat unexpectedly, go back to basics and review how points and vectors differ, and how each is represented. The need for this arises because, to represent a line or plane we must "add points together", and "scale points", operations that for points are nonsensical. To see what is really going on we introduce the notion of a coordinate frame, that makes clear the significant difference between a point and a vector, and reveals in what sense it is legitimate to "add points". The use of coordinate frames leads ultimately to the notion of "homogeneous coordinates", which is a central tool in computer graphics, and greatly simplifies many algorithms. We will make explicit use of coordinate frames in only a few places in the book, most notably when changing coordinate systems and "flying" cameras around a scene (see Chapters 5, 6, and 7) [4]. But even when not explicitly mentioned, an underlying coordinate frame will be present in every situation.

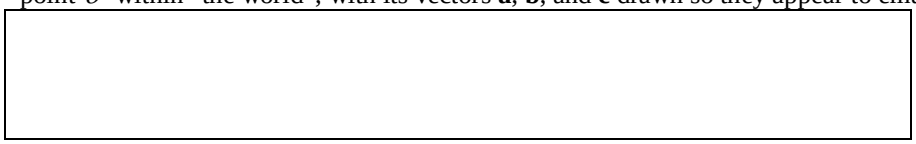
4.5.1. Coordinate Systems and Coordinate Frames.

One doesn't discover new lands without consenting
to lose sight of the shore for a very long time.
Andre Gide

When discussing vectors in previous sections we say, for instance, that a vector **v** = (3, 2, 7), meaning it is a certain 3-tuple. We say the same for a point, as in point *P* = (5, 3, 1). This makes it seem that points and vectors are the same thing. But points and vectors are very different creatures: points have location but no size or direction; vectors have size and direction but no location.

What we mean by **v** = (3, 2, 7), of course, is that the vector **v** has "components" (3, 2, 7) in the underlying coordinate system. Similarly, *P* = (5, 3, 1) means point *P* has coordinates (5, 3, 1) in the underlying coordinate system. Normally this confusion between the object and its representation presents no problem. The problem arises when there is more than one coordinate system (a very common occurrence in graphics), and when you transform points or vectors from one system into another.

We usually think of a coordinate system as three "axes" emanating from an origin, as in Figure 4.2b. But in fact a coordinate system is "located" somewhere in "the world", and its axes are best described by three vectors that point in mutually perpendicular directions. In particular it is important to make explicit the "location" of the coordinate system. So we extend the notion of a 3D coordinate system[5] to that of a 3D coordinate "frame." A **coordinate frame** consists of a specific point, $\mathcal{O}$, called the *origin*, and three mutually perpendicular unit vectors[6], **a**, **b**, and **c**. Figure 4.18 shows a coordinate frame "residing" at some point $\mathcal{O}$ within "the world", with its vectors **a**, **b**, and **c** drawn so they appear to emanate from $\mathcal{O}$ like axes.



[4] This is an area where graphics programmers can easily go astray: their programs produce pictures that look OK for simple situations, and become mysteriously and glaringly wrong when things get more complex.

[5] The ideas for a 2D system are essentially identical.

[6] In more general contexts the vectors need not be mutually perpendicular, but rather only "linearly independent" (such that, roughly, none of them is a linear combination of the other two). The coordinate frames we work with will always have perpendicular axis vectors.

Figure 4.18. A coordinate frame positioned in "the world".

Now to represent a vector **v** we find three numbers, (v_1, v_2, v_3) such that

$$\mathbf{v} = v_1\mathbf{a} + v_2\,\mathbf{b} + v_3\,\mathbf{c} \qquad (4.32)$$

and say that **v** "has the representation" (v_1, v_2, v_3) in this system.

On the other hand, to represent a point, *P*, we view its location as an offset from the origin by a certain amount: we represent the vector $P - \vartheta$ by finding three numbers (p_1, p_2, p_3) such that:

$$P - \vartheta = p_1\mathbf{a} + p_2\mathbf{b} + p_3\mathbf{c}$$

and then equivalently write *P* itself as:

$$P = \vartheta + p_1\mathbf{a} + p_2\mathbf{b} + p_3\mathbf{c} \qquad (4.33)$$

The representation of *P* is not just a 3-tuple, but a 3-tuple along with an origin. *P* is "at" a location that is offset from the origin by $p_1\mathbf{a} + p_2\mathbf{b} + p_3\mathbf{c}$. The basic idea is to make the origin of the coordinate system *explicit*. This becomes important only when there is more than one coordinate frame, and when transforming one frame into another.

Note that when we earlier defined the standard unit vectors **i**, **j**, and **k** as (1, 0, 0), (0, 1, 0), and (0, 0, 1), respectively, we were actually defining their *representations* in an underlying coordinate frame. Since by Equation 4.32 $\mathbf{i} = 1\mathbf{a} + 0\mathbf{b} + 0\mathbf{c}$, vector **i** is actually just **a** itself! It's a matter of naming: whether you are talking about the vector or about its representation in a coordinate frame. We usually don't bother to distinguish them.

Note that you can't explicitly say where ϑ is, or cite the directions of **a**, **b**, and **c**: To do so requires having some other coordinate frame in which to represent this one. In terms of its own coordinate frame, ϑ has the representation (0, 0, 0), **a** has the representation (1, 0, 0), etc.

The homogeneous representation of a point and a vector.

It is useful to represent both points and vectors using the *same* set of basic underlying objects, (**a**, **b**, **c**, ϑ). From Equations 4.32 and 4.33 the vector $\mathbf{v} = v_1\mathbf{a} + v_2\,\mathbf{b} + v_3\,\mathbf{c}$ then needs the four coefficients (v_1, v_2, v_3, 0) whereas the point $P = p_1\mathbf{a} + p_2\mathbf{b} + p_3\mathbf{c} + \vartheta$ needs the four coefficients (p_1, p_2, p_3, 1). The fourth component designates whether the object does or does not include ϑ. We can formally write any **v** and *P* using a matrix multiplication (multiplying a row vector by a column vector - see Appendix 2):

$$\mathbf{v} = (\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathcal{O}) \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{pmatrix} \tag{4.34}$$

$$P = (\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathcal{O}) \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{pmatrix} \tag{4.35}$$

Here the row matrix captures the nature of the coordinate frame, and the column vector captures the representation of the specific object of interest. Thus vectors and points have different representations: there is a fourth component of 0 for a vector and 1 for a point. This is often called the **homogeneous representation.**[7] The use of homogeneous coordinates is one of the hallmarks of computer graphics, as it helps to keep straight the distinction between points and vectors, and provides a compact notation when working with affine transformations. It pays off in a computer program to represent the points and vectors of interest in homogeneous coordinates as 4-tuples, by appending a 1 or 0[8]. This is particularly true when we must convert between one coordinate frame and another in which points and vectors are represented.

It is simple to convert between the "ordinary" representation of a point or vector (a 3-tuple for 3D objects or a 2-tuple for 2D objects) and the homogeneous form:

To go from ordinary to homogeneous coordinates:

if it's a point append a 1;
if it's a vector, append a 0;

To go from homogeneous coordinates to ordinary coordinates:

If it's a vector its final coordinate is 0. Delete the 0.
If it's a point its final coordinate is 1 Delete the 1.

OpenGL uses 4D homogeneous coordinates for all its vertices. If you send it a 3-tuple in the form (x, y, z), it converts it immediately to (x, y, z, 1). If you send it a 2D point (x, y), it first appends a 0 for the z-component and then a 1, to form (x, y, 0, 1). All computations are done within OpenGL in 4D homogeneous coordinates.

Linear Combinations of Vectors .
Note how nicely some things work out in homogeneous coordinates when we combine vectors coordinate-wise: all the definitions and manipulations are consistent:

• The difference of two points (x, y, z, 1) and (u, v, w, 1) is (x - u, y - v, z - w, 0), which is, as expected, a vector.
• The sum of a point (x, y, z, 1) and a vector (d, e, f, 0) is (x + d, y + e, z + f, 1), another point;
• Two vectors can be added: (d, e, f, 0) + (m, n, r, 0) = (d + m, e + n, f + r, 0) which produces another vector;
• It is meaningful to scale a vector: 3(d, e, f, 0) = (3d, 3e, 3f, 0);

[7] Actually we are only going part of the way in this discussion. As we see in Chapter 7 when studying projections, homogeneous coordinates in that context permit an additional operation, which makes them truly "homogeneous". Until we examine projections this operation need not be introduced.

[8] In the 2D case, points are 3-tuples (p_1, p_2, 1) and vectors are 3-tuples (v_1, v_2, 0).

• It is meaningful to form *any* linear combination of vectors. Let the vectors be $\mathbf{v} = (v_1, v_2, v_3, 0)$ and $\mathbf{w} = (w_1, w_2, w_3, 0)$. Then using arbitrary scalars a and b, we form $a\mathbf{v} + b\mathbf{w} = (av_1 + bw_1, av_2 + bw_2, av_3 + bw_3, 0)$, which is a legitimate vector.

Forming a linear combination of vectors is well defined, but does it make sense for points? The answer is no, except in one special case, as we explore next.

4.5.2. Affine Combinations of Points.

Consider forming a linear combination of two points, $P = (P_1, P_2, P_3, 1)$ and $R = (R_1, R_2, R_3, 1)$, using the scalars f and g:

$$fP + gR = (fP_1 + gR_1, fP_2 + gR_2, fP_3 + gR_3, f + g)$$

We know this is a legitimate vector if $f + g = 0$ (why?). But we shall see that it is *not* a legitimate point unless $f + g = 1$! Recall from Equation 4.2 that when the coefficients of a linear combination sum to 1 it is called an "affine" combination. So we see that the only linear combination of points that is legitimate is an affine combination. Thus, for example, the object $0.3P + 0.7R$ is a legitimate point, as are $2.7P - 1.7R$ and the midpoint $0.5P + 0.5R$, but $P + R$ is not a point. For three points, P, R, and Q we can form the legal point $0.3P + 0.9R - 0.2Q$, but not $P + Q - 0.9R$..

<u>Fact:</u> any affine combination of points is a legitimate point.

But what's wrong geometrically with forming *any* linear combination of two points, say

$$E = fP + gR \tag{4.36}$$

when $f + g$ is different from 1? The problem arises if we shift the origin of the coordinate system [Goldman85]. Suppose the origin is shifted by vector **u**, so that P is altered to $P + \mathbf{u}$ and R is shifted to $R + \mathbf{u}$. If E is a legitimate point, it too must be shifted to the new point $E' = E + \mathbf{u}$. But instead we have

$$E' = fP + gR + (f + g)\mathbf{u}$$

which is *not* $E + \mathbf{u}$ unless $f + g = 1$.

The failure of a simple sum $P_1 + P_2$ of two points to be a true point is shown in Figure 4.19. Points P_1 and P_2 are shown represented in two coordinate systems, one offset from the other. Viewing each point as the head of a vector bound to its origin, we see that the sum $P_1 + P_2$ yields two different points in the two systems. Therefore $P_1 + P_2$ depends on the choice of coordinate system. Note, by way of contrast, that the affine combination $0.5(P_1 + P_2)$ does *not* depend on this choice.

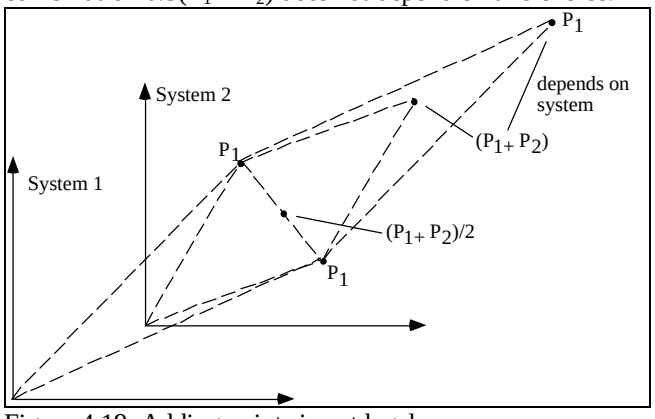


Figure 4.19. Adding points is not legal.

A Point plus a Vector is an Affine Combination of Points.

There is another way of examining affine sums of points that is interesting on its own, and also leads to a useful tool in graphics. It doesn't require the use of homogeneous coordinates.

Consider forming a point as a point A offset by a vector $\mathbf{v}$ that has been scaled by scalar t: $A + t\mathbf{v}$. This is the sum of a point and a vector so it is a legitimate point. If we take as vector $\mathbf{v}$ the difference between some other point B and A: $\mathbf{v} = B - A$ then we have the point P:

$$P = A + t(B - A) \tag{4.37}$$

which is also a legitimate point. But now rewrite it algebraically as:

$$P = tB + (1 - t)A \tag{4.38}$$

and it is seen to be an affine combination of points (why?). This further legitimizes writing affine sums of points. In fact, any affine sum of points can be written as a point plus a vector (see the exercises). If you are ever uncomfortable writing an affine sum of points as in Equation 4.38 (a form we will use often), simply understand that it *means* the point given by Equation 4.37.

Example 4.5.1: The centroid of a triangle. Consider the triangle T with vertices A, B, and C shown in Figure 4.20. We use the ideas above to show that the three **medians** of T meet at a point that lies 2/3 of the way along each median. This is the centroid (center of gravity[9]) of T.

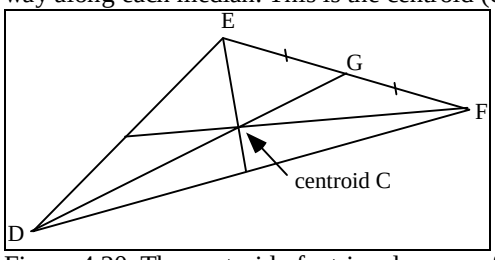


Figure 4.20. The centroid of a triangle as an affine combination.

By definition the median from D is the line from D to the midpoint of the opposite side. Thus $G = (E + F)/2$. We first ask where the point that is 2/3 of the way from D to G lies? Using the parametric form the desired point must be $D + (G - D)t$ with $t = 2/3$, which yields the affine combination C given by

$$C = \frac{D + E + F}{3}$$

(Try it!) Here's the cute part [pedoe70]. Since this result is *symmetrical* in D, E, and F, it must also be 2/3 of the way along the median from E, and 2/3 of the way along the median from F. Hence the 3 medians meet there, and C is the centroid.

This result generalizes nicely for a regular polygon of N sides: the centroid is simply the average of the N vertex locations, another affine combination. For an arbitrary polygon the formula is more complex

Practice Exercises.

4.5.1. Any affine combination of points is legitimate. Consider three scalars a, b, and c that sum to one, and three points A, B, and C. The affine combination $a\,A + b\,B + c\,C$ *is* a legal point because using $c = 1 - a - b$ it is seen to be the same as $a\,A + b\,B + (1 - a - b)\,C = C + a\,(A - C) + b\,(B - C)$, the sum of a point and two vectors (check this out!). To generalize: Given the affine combination of points $w_1A_1 + w_2A_2 + ... + w_nA_n$, where $w_1 + w_2 + ... + w_n = 1$, show that it can be written as a point plus a vector, and is therefore a legitimate point.

4.5.2. Shifting the coordinate system [Goldman85]. Consider the general situation of forming a linear combination of m points:

9The reference to gravity arises because if a thin plate is cut in the shape of T, the plate hangs level if suspended by a thread attached at the centroid. Gravity pulls equally on all sides of the centroid, so the plate is balanced.

$$E = \sum_{i=1}^{m} a_i P_i$$

We ask whether E is a point, a vector, or nothing at all? By considering the effect of a shift in each P_i by **u** show that E is "shifted" to $E' = E + S$ **u**, where S is the sum of the coefficients:

$$S = \sum_{i=1}^{m} a_i$$

Show that:
i). E is a point if $S = 1$.
ii). E is a vector if $S = 0$.
iii). E is meaningless for other values of S.

4.5.3. Linear Interpolation of two points.

The affine combination of points expressed in Equation 4.33:

$P = A(1 - t) + Bt$

performs **linear interpolation** between the points A and B. That is, the x-component $P_x(t)$ provides a value that is fraction t of the way between the value A_x and B_x, and similarly for the y-component (and in 3D the z-component). This is a sufficiently important operation to warrant a name, and *lerp*() (for linear interpolation) has become popular. In one dimension, *lerp*(a, b, t) provides a number that is the fraction t of the way from a to b. Figure 4.21 provides a simple implementation of `lerp()`.

```
float lerp(float a, float b, float t)
{
      return a + (b - a) * t;  // return a float
}
```

Figure 4.21. Linear interpolation effected by `lerp()`.

Similarly, one often wants to compute the point $P(t)$ that is fraction t of the way along the straight line from point A to point B. This point is often called the "tween" (for "in-between") at t of points A and B. Each component of the resulting point is formed as the `lerp`() of the corresponding components of A and B. A procedure

`Point2 canvas:: Tween(Point2 A, Point2 B, float t)` // tween A and B

is easily written (how?) to implement tweening. A 3D version is almost the same.

Example 4.5.2. Let $A = (4, 9)$ and $B = (3, 7)$. Then `Tween(A, B, t)` returns the point $(4 - t, 9 - 2t)$, so that `Tween`(A, B, 0.4) returns (3.6, 8.1). (Check this on graph paper.)

4.5.3. "Tweening" for Art and Animation.

Interesting animations can be created that show one figure being "tweened" into another. It's simplest if the two figures are polylines (or families of polylines) based on the same number of points. Suppose the first figure, A, is based on the polyline with points A_i , and the second polyline, B, is based on points B_i, for i = 0, . . . , n-1. We can form the polyline $P(t)$, called the "tween at t", by forming the points:

$P_i(t) = (1 - t) A_i + t B_i$

If we look at a succession of values for t between 0 and 1, say, t = 0, 0.1, 0.2, ..., 0.9, 1.0, we see that this polyline begins with the shape of A and ends with the shape of B, but in between it is a blend of the two

shapes. For small values of t it looks like *A*, but as t increases it warps (smoothly) towards a shape close to *B*. For $t = 0.25$, for instance, point $P_i(.25)$ of the tween is 25% of the way from *A* to *B*.

Figure 4.22 shows a simple example, in which polyline *A* has the shape of a house, and polyline *B* has the shape of the letter 'T'. The point *R* on the house corresponds to point *S* on the 'T'. The various tweens of point *R* on the house and point *S* on the T lie on the line between *R* and *S*. The tween for $t = 1/2$ lies at the midpoint of *RS*. The in between polylines show the shapes of the tweens for $t = 0, 0.25, 0.5, 0.75$, and 1.0.

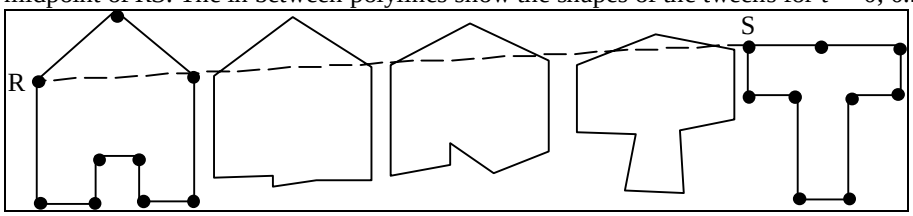


Figure 4.22. Tweening a "T" into a house.

Figure 4.23 shows `drawTween()`, that draws a tween of two polylines *A* and *B*, each having *n* vertices, at the specified value of *t*.

```
void canvas:: drawTween(Point2 A[], Point2 B[], int n, float
t)
{   // draw the tween at time t between polylines A and B
  for(int i = 0; i < n; i++)
  {
     Point2 P;
     P = Tween(A[i], B[i],t);
     if(i == 0)    moveTo(P.x, P.y);
     else          lineTo(P.x, P.y);
   }
}
```
Figure 4.23. Tweening two Polylines.

`drawTween()` could be used in an animation loop that tweens *A* and *B* back and forth, first as *t* increases from 0 to 1, then as *t* decreases back to 0, etc. Double buffering, as discussed in Chapter 3, is used to make the transition from one displayed tween to the next instantaneous.

```
for(t = 0.0, delT = 0.1; ; t += delT)  // tween back and forth forever
{
      <clear the buffer>
      drawTween(A, B, n, t);
      glutSwapBuffers();
      if( t >= 1.0 || t <= 0.0) delT = - delT; // reverse the flow of t
}
```

Figure 4.24 shows an artistic use of this technique based on two sets of polylines. Three tweens are shown (what values of *t* are used?). Because the two sets of polylines are drawn sufficiently far apart, there is room to draw the tweens between them with no overlap, so that all five pictures fit nicely on one frame.

see Figure 7.11 from first edition

Figure 4.24. From man to woman. (Courtesy of Marc Infield.)

Susan E. Brennan of Hewlett Packard in Palo Alto, California, has produced caricatures of famous figures using this method (see [dewdney88]). Figure 4.25 shows an example. The second and fourth faces are based on digitized points for Elizabeth Taylor and John F. Kennedy. The third face is a tween, and the other three are based on **extrapolation**. That is, values of *t* larger than 1 are used, so that the term $(1 - t)$ is negative. Extrapolation can produce caricature-like distortions, in some sense "going to the other side" of polyline *B* from polyline *A*. Values of *t* less than 0 may also be used, with a similar effect.

see Figure 7.12 from 1st edition: Elizabeth Taylor to J.F. Kennedy

Figure 4.25. Face Caricature: Tweening and extrapolation. (Courtesy of Susan Brennan.)

Tweening is used in the film industry to reduce the cost of producing animations such as cartoons. In earlier days an artist had to draw 24 pictures for each second of film, because movies display 24 frames per second. With the assistance of a computer, however, an artist need draw only the first and final pictures, called **key-frames**, in certain sequences and let the others be generated automatically. For instance, if the characters are not moving too rapidly in a certain one-half-second portion of a cartoon, the artist can draw and digitize the first and final frames of this portion, and the computer can create 10 tweens using linear interpolation, thereby saving a great deal of the artist's time. See the case study at the end of this chapter for a programming project that produces these effects.

Practice Exercises.

4.5.3. A Limiting Case of Tweening. What is the effect of tweening when all of the points A_i in polyline A are the same? How is polyline B distorted in its appearance in each tween?

4.5.4. An Extrapolation. Polyline A is a square with vertices (1, 1), (-1, 1), (-1, -1), (1, -1), and polyline B is a wedge with vertices (4, 3), (5, -2), (4, 0), (3, -2). Sketch (by hand) the shape $P(t)$ for t = -1, -0.5, 0.5, and 1.5.

4.5.5. Extrapolation Versus Tweening. Suppose that five polyline pictures are displayed side by side. From careful measurement you determine that the middle three are in-betweens of the first and the last, and you calculate the values of t used. But someone claims that the last is actually an extrapolation of the first and the fourth. Is there any way to tell whether this is true? If it is an extrapolation, can the value of t used be determined? If so, what is it?

4.5.4. Preview: Quadratic and cubic tweening, and Bezier Curves.

In Chapter 8 we address the problem of designing complex shapes called Bezier curves. It is interesting to note here that the underlying idea is simply tweening between a collection of points. With linear interpolation above we "partition unity" into the pieces (1 - t) and t, and use these pieces to "weight" the points A and B. We can extend this to quadratic interpolation by partitioning unity into three pieces. Just rewrite 1 as

$$1 = ((1-t) + t)^2$$

and expand it to produce the three pieces $(1 - t)^2$, $2(1 - t)\, t$, and t^2. They obviously sum to one, so they can be used to form the affine combination of points A, B, and C:

$$P(t) = (1 - t)^2 A + 2(1 - t) B + t C$$

This is the "Bezier curve" for the points A, B, and C. Figure 4.26a shows the shape of $P(t)$ as t varies from 0 to 1. It flows smoothly from A to C. (Notice that the curve misses the middle point.) Going further, one can expand $((1 - t) + t)^3$ into four pieces (which ones?) which can be used to do "cubic interpolation" between four points A, B, C and D, as shown in Figure 4.26b.

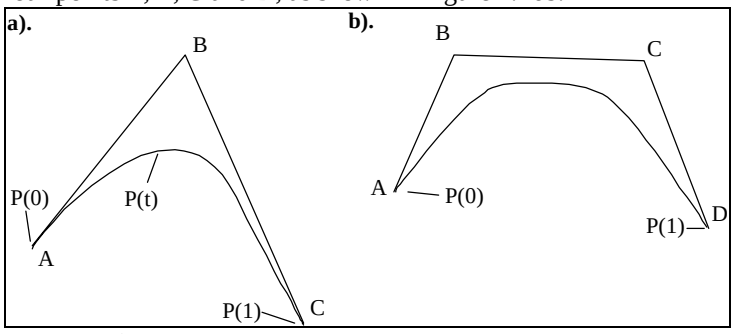


Figure 4.26. Bezier curves as Tweening.

Practice Exercise 4.5.6. Try it out. Draw three points *A*, *B*, and *C* on a piece of graph paper. For each of the values $t = 0, .1, .2, ..., .9, 1$ compute the position of $P(t)$ in Equation 4.38, and draw the polyline that passes through these points. Is it always a parabola?

4.5.5. Representing Lines and Planes.

We now turn to developing the principal forms in which lines and planes are represented mathematically. It is quite common to find data structures within a graphics program that capture a line or plane using one of these forms.

Lines in 2D and 3D space.

A **line** is defined by two points, say *C* and *B* (see Figure 4.27a). It is infinite in length, passing through the points and extending forever in both directions. A **line segment** (**segment** for short) is also defined by two points, its **endpoints,** but extends only from one endpoint to the other (Figure 4.27b). Its **parent** line is the infinite line that passes through its endpoints. A **ray** is "semi-infinite." It is specified by a point and a direction. It "starts" at a point and extends infinitely far in a given direction (Figure 4.27c).

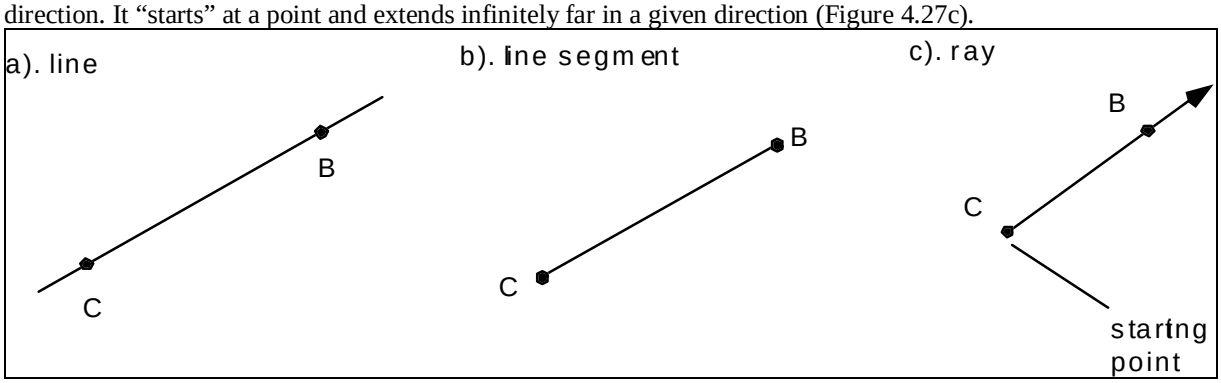


Figure 4.27. Lines, segments, and rays.

These objects are very familiar, yet it is useful to collect their important representations and properties in one spot. We also describe the most important representation of all for a line in computer graphics, the **parametric representation.**

The parametric representation of a line.

The construction in Equations 4.32 and 4.33 is very useful, because as *t* varies the point *P* traces out all of the points on the straight line defined by *C* and *B*. The construction therefore gives us a way to name and compute any point along this line.

This is done using a **parameter** *t* that distinguishes one point on the line from another. Call the line *L*, and give the name $L(t)$ to the position associated with *t*. Using $\mathbf{b} = B - C$ we have:

$$L(t) = C + \mathbf{b}\, t \qquad (4.39)$$

As *t* varies so does the position of $L(t)$ along the line. (One often thinks of *t* as "time", and uses language such as: "at time 0 ...", "as time goes on..", or "later" to describe different parts of the line.) Figure 4.28 shows vector **b** and the line *L* passing through *C* and *B*. (A 2D version is shown but the 3D version uses the same ideas.) Note where $L(t)$ is located for various values of *t*. If $t = 0$, $L(0)$ evaluates to *C* so at $t = 0$ we are "at" point *C*. At $t = 1$ then $L(1) = C + (B - C) = B$. As *t* varies we add a longer or shorter version of **b** to the point *C*, resulting in a new point along the line. If *t* is larger than 1 this point lies somewhere on the opposite side of *C* from *B*, and when *t* is less than 0 it lies on the side of *C* opposite from *B*.

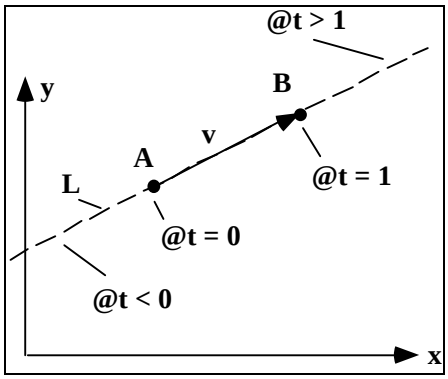


Figure 4.28. Parametric representation $L(t)$ of a line.

For a fixed value of t, say $t = 0.6$, Equation 4.39 gives a formula for exactly one point along the line through C and B: the particular point $L(0.6)$. Thus it is a description of a point. But since one can view it as a function of t that generates the coordinates of *every* point on L as t varies, it is called the **parametric representation of line *L*.**

The line, ray, and segment of Figure 4.26 are all represented by the same $L(t)$ of Equation 4.39. They differ parametrically only in the values of t that are relevant:

segment : $0 \le t \le 1$
ray: $0 \le t < \infty$ (4.40)
line : $-\infty < t < \infty$

The ray "starts" at C when $t = 0$ and passes through B at $t = 1$, then continues forever as t increases. C is often called the "starting point" of the ray.

A very useful fact is that $L(t)$ lies "fraction t of the way" between C and B when t lies between 0 and 1. For instance, when $t = 1/2$ the point $L(0.5)$ is the **midpoint** between C and B, and when $t = 0.3$ the point $L(0.3)$ is 30% of the way from C to B. This is clear from Equation 4.39 since $|L(t) - C = |\mathbf{b}|\,|t|$ and $|B - C| = |\mathbf{b}|$, so the value of |t| is the ratio of the distances $|L(t) - C|$ to $|B - C|$, as claimed.

One can also speak of the "speed" with which the point $L(t)$ "moves" along line L. Since it covers distance $|\mathbf{b}|\,t$ in time t it is moving at constant speed $|\mathbf{b}|$.

Example 4.5.2. A line in 2D. Find a parametric form for the line that passes through C= (3, 5) and B = (2, 7). **Solution:** Build vector $\mathbf{b} = B - C$ = (-1, 2) to obtain the parametric form L(t) = (3 - t, 2 + 2 t).
Example 4.5.3. A line in 3D. Find a parametric form for the line that passes through C= (3, 5,6) and B = (2, 7,3). **Solution:** Build vector $\mathbf{b} = B - C$ = (-1, 2, -3) to obtain the parametric form L(t) = (3 - t, 2 + 2 t, 6 - 3t).

Other parametrizations for a straight line are possible, although they are rarely used. For instance, the point $W(t)$ given by

$$W(t) = C + \mathbf{b}t^3$$

also "sweeps" over every point on L. It lies at C when $t = 0$, and reaches B when $t = 1$. Unlike $L(t)$, however, W(t) "accelerates" along its path from C to B.

Point normal form for a line (the implicit form).
This is the same as the equation for a line, but we rewrite it in a way that better reveals the underlying geometry. The familiar equation of a line in 2D has the form

$$f x + g y = 1 \qquad (4.41)$$

where f and g are some constants. The notion is that every point (x, y) that satisfies this equation lies on the line, so it provides a condition for a point to be on the line. Note: This is true only for a line in 2D; a line in 3D requires two equations. So, unlike the parametric form that works perfectly well in 2D and 3D, the point normal form only applies to lines in 2D.

This equation can be written using a dot product: $(f, g) \cdot (x, y) = 1$, so for every point on a line a certain dot product must have the same value. We examine the geometric interpretation of the "vector" (f, g), and in so doing develop the "point normal" form of a line. It is very useful in such tasks as clipping, hidden line elimination, and ray tracing. Formally the point normal form makes no mention of dimensionality: A line in 2D has a point normal form, and a plane in 3D has one.

Suppose that we know line L passes through points C and B, as in Figure 4.29. What is its point normal form? If we can find a vector $\mathbf{n}$ that is perpendicular to the line, then for any point $R = (x, y)$ on the line the vector $R - C$ must be perpendicular to $\mathbf{n}$, so we have the condition on R:

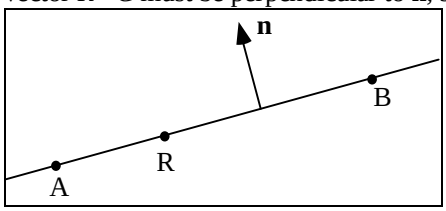


Figure 4.29. Finding the point normal form for a line.

$$\mathbf{n} \cdot (R - C) = 0 \qquad \text{(point normal form)} \qquad (4.42)$$

This is the **point normal** equation for the line, expressing that a certain dot product must turn out to be zero for *every* point R on the line. It employs as data *any* point lying on the line, and *any* normal vector to the line.

We still must find a suitable **n.** Let $\mathbf{b} = B - C$ denote the vector from C to B. Then $\mathbf{b}^{\perp}$ will serve well as the desired **n**. For purposes of building the point normal form, any scalar multiple of $\mathbf{b}^{\perp}$ works just as well for **n**.

Example 4.5.4. Find the point normal form. Suppose line L passes through points $C = (3, 4)$ and $B = (5, -2)$. Then $\mathbf{b} = B - C = (2, -6)$ and $\mathbf{b}^{\perp} = (6, 2)$ (sketch this). Choosing C as the point on the line, the point normal form is: $(6, 2) \cdot ((x, y) - (3, 4)) = 0$, or $6x + 2y = 26$. Both sides of the equation can be divided by 26 (or any other nonzero number) if desired.

It's also easy to find the normal to a line given the equation of the line, say, $f x + g y = 1$. Writing this once again as $(f, g) \cdot (x, y) = 1$ it is clear that the normal $\mathbf{n}$ is simply (f, g) (or any multiple thereof). For instance, the line given by $5x - 2y = 7$ has normal vector $(5, -2)$, or more generally $K(5, -2)$ for any nonzero K.

It's also straightforward to find the parametric form for a line if you are given its point normal form. Suppose it is known that line L has point normal form $\mathbf{n} \cdot (P - C) = 0$, where $\mathbf{n}$ and C are given explicitly. The parametric form is then $L(t) = C + \mathbf{n}^{\perp} t$ (why?). You can also obtain the parametric form if the equation of the line is given. a). find the normal $\mathbf{n}$ as in the previous paragraph, and b). find a point (C_x, C_y) on the line by choosing any value for C_x and use the equation to find the corresponding C_y.

Moving from each representation to the others.
We have described three different ways to characterize a line. Each representation uses certain data that distinguishes one line from another. This is the data that would be stored in a suitable data structure within a program to capture the specifics of each line being stored. For instance, the data associated with the representation that specifies a line parametrically as in $C + \mathbf{b}t$ would be the point C and the direction $\mathbf{b}$. We summarize this by saying the relevant data is $\{C, \mathbf{b}\}$.

The three representations and their data are:

- The two point form: say C and B; data = $\{C, B\}$
- The parametric form: $C + \mathbf{b}t$; data = $\{C, \mathbf{b}\}$.
- The point normal (implicit) form (in 2D only): $\mathbf{n} \cdot (P - C) = 0$; data = $\{C, \mathbf{n}\}$.

Note that a point C on the line is common to all three forms. Figure 4.30 shows how the data in each representation can be obtained from the data in the other representations. For instance, given $\{C, \mathbf{b}\}$ of the parametric form, the normal $\mathbf{n}$ of the point normal form is obtained simply as $\mathbf{b}^{\perp}$.

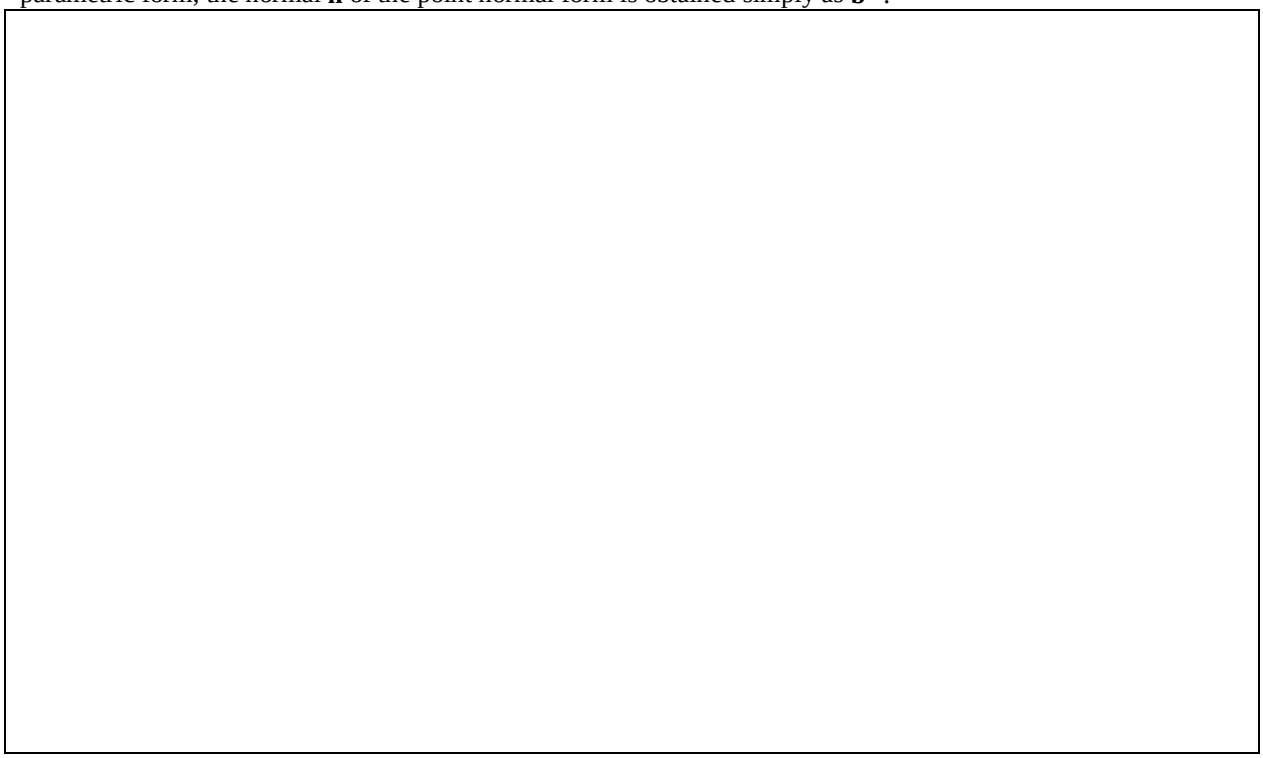


Figure 4.30. Moving between representations of a line.

Practice Exercise 4.5.5. Find the point normal form. Find the point normal form for the line that passes through (-3, 4) and (6, -1). Sketch the line and its normal vector on graph paper.

Planes in 3D space.

Because there is such a heavy use of polygons in 3D graphics, planes seem to appear everywhere. A polygon (a "face"of an object) lies in its "parent" plane, and we often need to clip objects against planes, or find the plane in which a certain face lies.

Planes, like lines, have three fundamental forms: the three-point form, the parametric representation and the point normal form. We examined the three-point form in Section 4.4.2.

The parametric representation of a plane.

The parametric form for a plane is built on three ingredients: one of its points, C, and two (nonparallel) vectors, $\mathbf{a}$ and $\mathbf{b}$, that lie in the plane, as shown in Figure 4.31. If we are given the three (non-collinear) points A, B, and C in the plane, then take $\mathbf{a} = A - C$ and $\mathbf{b} = B - C$.

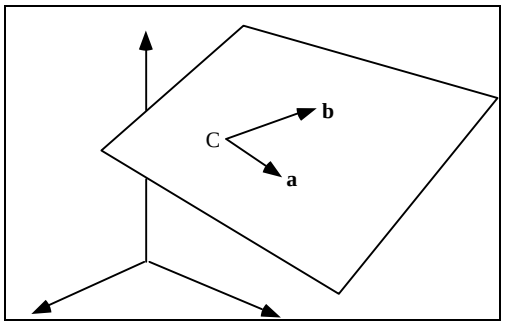


Figure 4.31. Defining a plane parametrically.

To construct a parametric form for this plane, note that any point in the plane can be represented by a vector sum: C plus some multiple of **a** plus some multiple of **b**. Using parameters s and t to specify the "multiples" we have $C + s\,\mathbf{a} + t\,\mathbf{b}$. This provides the desired parametric form $P(s, t)$

$$P(s, t) = C + \mathbf{a}\, s + \mathbf{b}\, t \qquad (4.43)$$

Given any values of s and t we can identify the corresponding point on the plane. For example, the position "at" $s = t = 0$ is C itself, and that at $s = 1$ and $t = -2$ is $P(1, -2) = C + \mathbf{a} - 2\,\mathbf{b}$.

Note that two parameters are involved in the parametric expression for a surface, whereas only one parameter is needed for a curve. In fact if one of the parameters is fixed, say $s = 3$, then $P(3, t)$ is a function of one variable and represents a straight line: $P(3, t) = (C + 3\,\mathbf{a}) + \mathbf{b}\, t$.

It is sometimes handy to arrange the parametric form into its "component" form by collecting terms

$$P(s, t) = (C_X + a_X s + b_X t,\ C_Y + a_Y s + b_Y t,\ C_Z + a_Z s + b_Z t). \qquad (4.44)$$

We can rewrite the parametric form in Equation 4.43 explicitly in terms of the given points A, B, and C: just use the definitions of **a** and **b**:

$$P(s, t) = C + s(A - C) + t(B - C)$$

which can be rearranged into the *affine combination* of points:

$$P(s, t) = s\,A + t\,B + (1 - s - t)C \qquad (4.45)$$

Example 4.5.6. Find a parametric form given three points in a plane. Consider the plane passing through $A = (3,3,3)$, $B = (5,5,7)$, and $C = (1, 2, 4)$. From Equation 4.43 it has parametric form $P(s, t) = (1, 2, 4) + (2, 1, -1)\, s + (4, 3, 3)\, t$. This can be rearranged to the component form: $P(s, t) = (1 + 2s + 4t)\ \mathbf{i} + (2 + s + 3t)\ \mathbf{j} + (4 - s + 3t)\ \mathbf{k}$, or to the affine combination form $P(s, t) = s(3, 3, 3) + t(5, 5, 7) + (1 - s - t)(1, 2, 4)$.

The point normal form for a plane.
Planes can also be represented in point normal form, and the classic equation for a plane emerges at once.

Figure 4.32 shows a portion of plane P in three dimensions. A plane is completely specified by giving a single point, $B = (b_X, b_Y, b_Z)$, that lies within it, and the normal direction, $\mathbf{n} = (n_X, n_Y, n_Z)$, to the plane. Just as the normal vector to a line in two dimensions orients the line, the normal to a plane orients the plane in space.

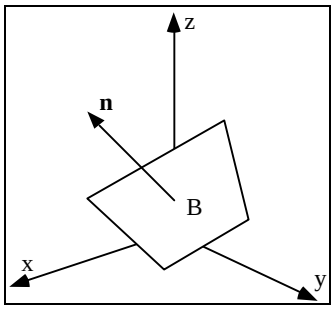


Figure 4.32. Determining the equation of a plane.

The normal **n** is understood to be perpendicular to any line lying in the plane. For an arbitrary point $R = (x, y, z)$ in the plane, the vector from R to B must be perpendicular to **n**, giving:

$$\mathbf{n} \cdot (R - B) = 0 \qquad (4.46)$$

This is the point normal equation of the plane. It is identical in form to that for the line: a dot product set equal to 0. All points in a plane form vectors with B that have the same dot product with the normal vector. By spelling out the dot product and using $\mathbf{n} = (n_X, n_Y, n_Z)$, we see that the point normal form is the traditional equation for a plane:

$$n_X x + n_Y y + n_Z z = D \qquad (4.47)$$

where $D = \mathbf{n} \cdot (B - 0)$. For example, if given the equation for a plane such as $5x - 2y + 8z = 2$, you know immediately that the normal to this plane is (5, -2, 8) or any multiple of this. (How do you find a point in this plane?)

Example 4.5.7. Find a point normal form. Let plane P pass through (1, 2, 3) with normal vector (2, -1, -2).
Its point normal form is $(2, -1, -2) \cdot ((x, y, z) - (1, 2, 3)) = 0$. The equation for the plane may be written out as $2x - y - 2z = 6$.

Example 4.5.8. Find a parametric form given the equation of the plane. Find a parametric form for the plane $2x - y + 3z = 8$. **Solution**: By inspection the normal is (2, - 1, 3). There are many parametrizations; we need only find one. For C, choose any point that satisfies the equation; $C =$ (4, 0, 0) will do. Find two (noncollinear) vectors, each having a dot product of 0 with (2, - 1, 3); some hunting finds that $\mathbf{a} =$ (1, 5, 1) and $\mathbf{b} =$ (0, 3, 1) will work. Thus the plane has parametric form $P(s, t) =$ (4, 0, 0) + (1, 5, 1) s + (0, 3, 1) t.

Example 4.5.9. Finding two noncollinear vectors. Given the normal **n** to a plane, what is an easy way to find two noncollinear vectors **a** and **b** that are both perpendicular to **n**? (In the previous exercise we just invented two that work.) Here we use the fact that the cross product of *any* vector with **n** is normal to **n**. So we take a simple choice such as (0, 0, 1), and construct **a** as its cross product with **n**:

$$\mathbf{a} = (0, 0, 1) \times \mathbf{n} = (-n_Y, n_X, 0)$$

(Is this indeed normal to **n**?). We can use the same idea to form **b** that is normal to both **n** and **a**:

$$\mathbf{b} = \mathbf{n} \times \mathbf{a} = (-n_X n_Z, -n_Y n_Z, n_X^2 + n_Y^2)$$

(Check that $\mathbf{b} \perp \mathbf{a}$ and $\mathbf{b} \perp \mathbf{n}$.) So **b** is certainly not collinear with **a**.

We apply this method to the plane $(3, 2, 5) \cdot (R - (2,7,0)) = 0$. Set $\mathbf{a} = (0, 0, 1) \times \mathbf{n} = (-2, 3, 0)$ and $\mathbf{b} = (-15, -10, 13)$. The plane therefore has parametric form:

$$P(s, t) = (2 - 2s - 15t, 7 + 3s - 10t, 13t).$$

Check: Is $P(s, t) - C = (-2s - 15t, -3s - 10t, 13t)$ indeed normal to **n** for *every* s and t?

Practice Exercise 4.5.7. Find the Plane. Find a parametric form for the plane coincident with the y, z-plane.

Moving from each representation to the others.
Just as with lines, it is useful to be able to move between the three representations of a plane, to manipulate the data that describes a plane into the form best suited to a problem.

For a plane, the three representations and their data are:

- The three point form: say C , B, and A; data = {C, B, A}
- The parametric form: $C + \mathbf{a}s + \mathbf{b}t$; data = {$C$, **a**, **b**}.
- The point normal (implicit) form: $\mathbf{n} \cdot (P - C) = 0$; data = {$C$, **n**}.

A point C on the plane is common to all three forms. Figure 4.33 shows how the data in each representation can be obtained from the data in the other representations. Check each one carefully. Most of these cases have been developed explicitly in Section 4.4.2 and this section. Some are developed in the exercises. The trickiest is probably the calculation in Example 4.5.10. Another that deserves some explanation is finding three points in a plane when given the point normal form. One point, C, is already known. The other two are found using special values in the point normal form itself, which is the equation $n_x x + n_y y + n_z z = \mathbf{n} \cdot C$. Choose ,for convenience, $A = (0, 0, a_z)$, and use the equation to determine that $a_z = \mathbf{n} \cdot C / n_z$. Similarly, choose $B = (0, b_y, 0)$, and use the equation to find $b_y = \mathbf{n} \cdot C / n_y$.

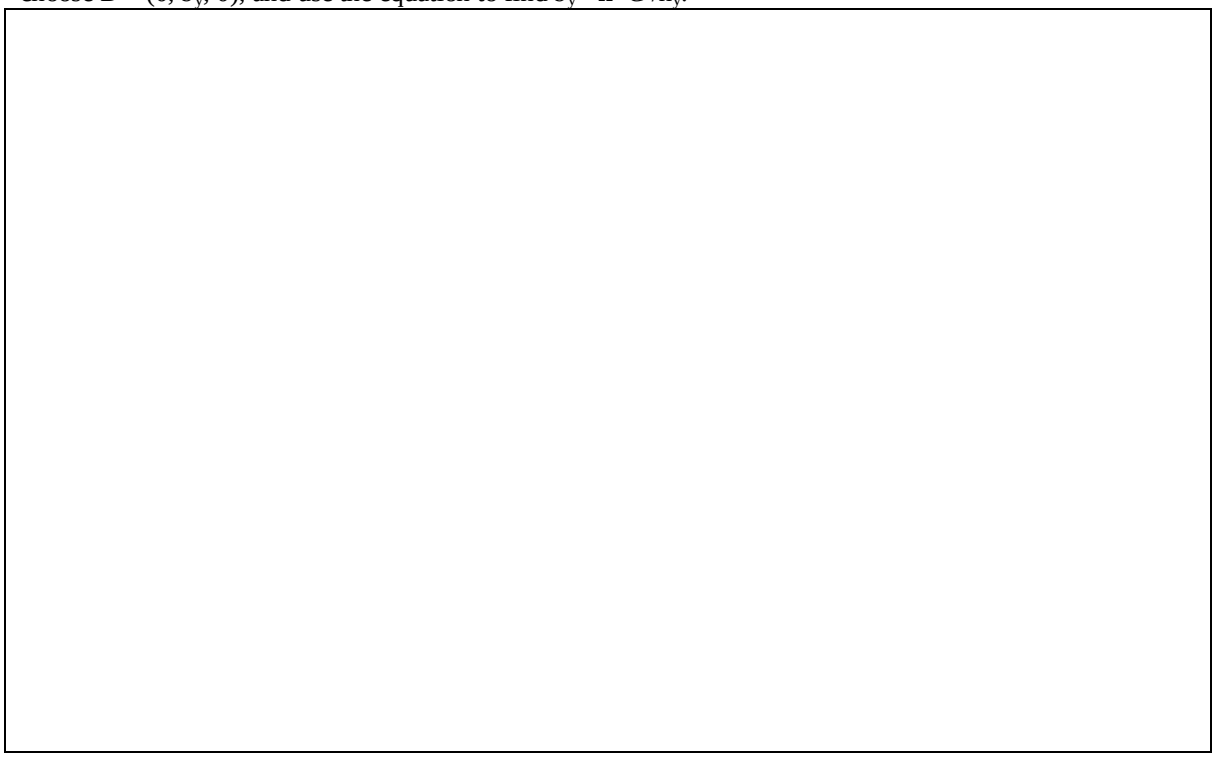


Figure 4.33. Moving between representations of a plane.

Planar Patches.

Just as we can restrict the parameter t in the representation of a line to obtain a ray or a segment, we can restrict the parameters s and t in the representation of a plane.

In the parametric form of Equation 4.43 the values for s and t can range from $-\infty$ to ∞, and thus the plane can extend forever. In some situations we want to deal with only a "piece" of a plane, such as a parallelogram that lies in it. Such a piece is called a **planar patch**, a term that invites us to imagine the plane as a quilt of many patches joined together. Later we examine curved surfaces made up of patches which are not necessarily planar. Much of the practice of modeling solids involves piecing together patches of various shapes to form the skin of an object.

A planar patch is formed by restricting the range of allowable parameter values for *s* and *t*. For instance, one often restricts *s* and *t* to lie only between 0 and 1. The patch is positioned and oriented in space by appropriate choices of **a**, **b**, and *C*. Figure 4.34a shows the available range of *s* and *t* as a square in **parameter space**, and Figure 4.34b shows the patch that results from this restriction in object space.

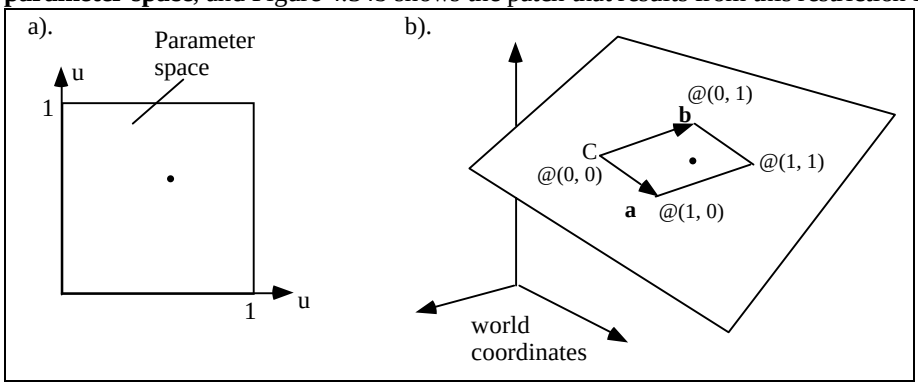


Figure 4.34. Mapping between two spaces to define a planar patch.

To each point (*s*, *t*) in parameter space there corresponds one 3D point in the patch $P(s, t) = C + \mathbf{a}s + \mathbf{b}t$. The patch is a parallelogram whose corners correspond to the four corners of parameter space and are situated at

$$\begin{aligned} P(0, 0) &= C; \\ P(1, 0) &= C + \mathbf{a}; \\ P(0, 1) &= C + \mathbf{b}; \\ P(1, 1) &= C + \mathbf{a} + \mathbf{b}. \end{aligned} \qquad (4.48)$$

The vectors **a** and **b** determine both the size and the orientation of the patch. If **a** and **b** are perpendicular, the grid will become rectangular, and if in addition **a** and **b** have the same length, the grid will become square. Changing *C* just shifts the patch without changing its shape or orientation.

Example 4.5.10. Make a patch. Let $C = (1, 3, 2)$, $\mathbf{a} = (1, 1, 0)$, and $\mathbf{b} = (1, 4, 2)$. Find the corners of the planar patch. **Solution**: From the preceding table we obtain the four corners: $P(0, 0) = (1, 3, 2)$, $P(0, 1) = (2, 7, 4)$, $P(1, 0) = (2, 4, 2)$, and $P(1, 1) = (3, 8, 4)$.

Example 4.5.11. Characterize a Patch. Find **a**, **b,** and *C* that create a square patch of length 4 on a side centered at the origin and parallel to the *x*, *z*-plane. **Solution:** The corners of the patch are at (2, 0, 2), (2, 0, - 2), (- 2, 0, 2), and (- 2, 0, - 2). Choose any corner, say (2, 0, - 2), for *C*. Then **a** and **b** each have length 4 and are parallel to either the *x*- or the *z*-axis. Choose $\mathbf{a} = (-4, 0, 0)$ and $\mathbf{b} = (0, 0, 4)$.

Practice Exercise 4.5.8. Find a Patch. Find point *C* and some vectors **a** and **b** that create a patch having the four corners (- 4, 2, 1), (1, 7, 4), (- 2, - 2, 2), and (3, 3, 5).

4.6. Finding the Intersection of two Line Segments.

We often need to compute where two line segments in 2D space intersect. It appears in many other tasks, such as determining whether or not a polygon is simple. Its solution will illustrate the power of parametric forms and dot products.

The Problem: Given two line segments, determine whether they intersect, and if they do, find their point of intersection.

Suppose one segment has endpoints *A* and *B* and the other segment has endpoints C and D. As shown in Figure 4.35 the two segments can be situated in many different ways: They can miss each other (a and b), overlap in one point (c and d), or even overlap over some region (e). They may or may not be parallel. We need an organized approach that handles all of these possibilities.

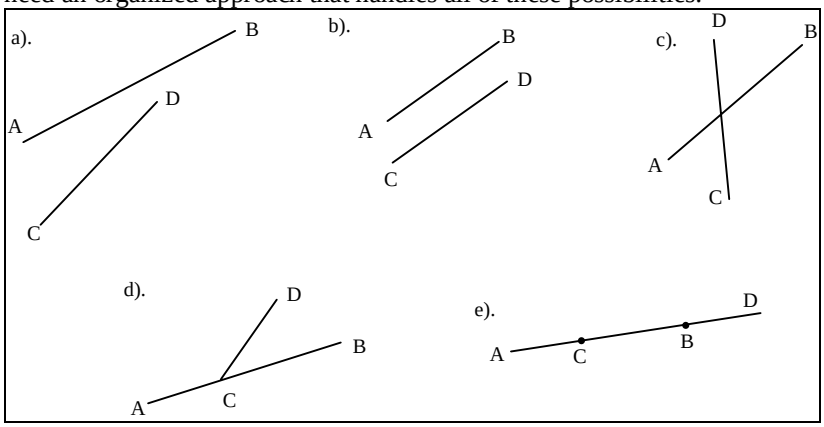


Figure 4.35. Many cases for two line segments.

Every line segment has a **parent line**, the infinite line of which it is part. Unless two parent lines are parallel they will intersect at some point. We first locate this point.

We set up parametric representations for each of the line segments in question. Call *AB* the segment from *A* to *B*. Then

$$AB(t) = A + \mathbf{b}\,t \qquad (4.49)$$

where for convenience we define $\mathbf{b} = B - A$. As *t* varies from 0 to 1 all points on the finite line segment are visited. If *t* is allowed to vary from $-\infty$ to ∞ the entire parent line is swept out.

Similarly we call the segment from *C* to *D* by the name *CD*, and give it parametric representation (using a new parameter, say, *u*)

$$CD(u) = C + \mathbf{d}\,u,$$

where $\mathbf{d} = D - C$. We use different parameters for the two lines, *t* for one and *u* for the other, in order to describe different points on the two lines independently. (If the same parameter were used, the points on the two lines would be locked together.)

For the parent lines to intersect, there must be specific values of *t* and *u* for which the two equations above are equal:

$$A + \mathbf{b}t = C + \mathbf{d}u$$

Defining $\mathbf{c} = C - A$ for convenience we can write this condition in terms of three known vectors and two (unknown) parameter values:

$$\mathbf{b}t = \mathbf{c} + \mathbf{d}u \qquad (4.50)$$

This provides two equations in two unknowns, similar to Equation 4.22. We solve it the same way: dot both sides with $\mathbf{d}^{\perp}$ to eliminate the term in *d*, giving $\mathbf{d}^{\perp} \cdot \mathbf{b}t = \mathbf{d}^{\perp} \cdot \mathbf{c}$. There are two main cases: the term $\mathbf{d}^{\perp} \cdot \mathbf{b}$ is zero or it is not.

Case 1: *The term* $\mathbf{d}^{\perp} \cdot \mathbf{b}$ *is not Zero.*

Here we can solve for *t* obtaining:

$$t = \frac{\mathbf{d}^{\perp} \cdot \mathbf{c}}{\mathbf{d}^{\perp} \cdot \mathbf{b}} \qquad (4.51)$$

Similarly "dot" both sides of Equation 4.50 with $\mathbf{b}^{\perp}$ to obtain (after using one additional property of perp-dot products—which one?):

$$u = \frac{\mathbf{b}^{\perp} \cdot \mathbf{c}}{\mathbf{d}^{\perp} \cdot \mathbf{b}} \qquad (4.52)$$

Now we know that the two parent lines intersect, and we know where. But this doesn't mean that the line segments themselves intersect. If *t* lies outside the interval [0, 1], segment *AB* doesn't "reach" the other segment, with similar statements if *u* lies outside of [0,1]. If both *t* and *u* lie between 0 and 1 the line segments *do* intersect at some point, *I*. The location of *I* is easily found by substituting the value of *t* in Equation 4.49:

$$I = A + \left(\frac{\mathbf{d}^{\perp} \cdot \mathbf{c}}{\mathbf{d}^{\perp} \cdot \mathbf{b}} \right) \mathbf{b} \qquad \text{(the intersection point)} \qquad (4.53)$$

Example 4.6.1: Given the endpoints *A* = (0, 6), *B* = (6, 1), *C* = (1, 3), and *D* = (5, 5), find the intersection if it exists. **Solution:** $\mathbf{d}^{\perp} \cdot \mathbf{b}$ = -32, so *t* = 7/16 and u = 13/32 which both lie between 0 and 1, and so the segments do intersect. The intersection lies at (*x*, *y*) = (21/8, 61/16). This result may be confirmed visually by drawing the segments on graph paper and measuring the observed intersection.

Case 2: *The term* $\mathbf{d}^{\perp} \cdot \mathbf{b}$ *is Zero.*

In this case we know **d** and **b** are parallel (why?). The segments might still overlap, but this can happen only if the parallel parent lines are identical. A test for this is developed in the exercises.

The exercises discuss developing a routine that performs the complete intersection test on two line segments.

Practice Exercises.

4.6.1. When the parent lines overlap. We explore case 2 above, where the term $\mathbf{d}^{\perp} \cdot \mathbf{b}$ = 0, so the parent lines are parallel. We must determine whether the parent lines are identical, and if so whether the segments themselves overlap.

To test whether the parent lines are the same, see whether *C* lies on the parent line through *A* and *B*.
a). Show that the equation for this parent line is $b_X (y - A_Y) - b_Y (x - A_X) = 0$.
We then substitute C_X for *x* and C_Y for *y* and see whether the left-hand side is sufficiently close to zero (i.e. its size is less than some tolerance such as 10^{-8}). If not, the parent lines do not coincide, and no intersection exists. If the parents lines are the same, the final test is to see whether the segments themselves overlap.
b). To do this, show how to find the two values t_C and t_d at which this line through A and B reaches *C* and *D*, respectively. Because the parent lines are identical, we can use just the *x*-component. Segment *AB* begins at 0 and ends at 1, and by examining the ordering of the four values 0, 1, t_c, and t_d, we can readily determine the relative positions of the two lines.
c). Show that there is an overlap unless both t_c and t_d are less than 0 or both are larger than 1. If there is an overlap, the endpoints of the overlap can easily be found from the values of t_C and t_d.
d). Given the endpoints *A* = (0, 6), *B* = (6, 2), *C* = (3, 4), and *D* = (9, 0), determine the nature of any intersection.

4.6.2. The Algorithm for determining the intersection. Write the routine `segIntersect`() that would be used in the context: `if(segIntersect(A, B, C, D, InterPt)) <do something>` It takes four points representing the two segments, and returns 0 if the segments do not intersect, and 1 if they do. If they do intersect the location of the intersection is placed in `interPt`. It returns -1 if the parent lines are identical.

4.6.3. Testing the Simplicity of a Polygon. Recall that a polygon *P* is simple if there are no edge intersections except at the endpoints of adjacent edges. Fashion a routine `int isSimple(Polygon P)` that takes a brute force approach and tests whether any pair of edges of the list of vertices of the polygon intersect, returning `0` if so, and 1 if not so. (`Polygon` is some suitable class for describing a polygon.) This is a simple algorithm but not the most efficient one. See [moret91] and [preparata85] for more elaborate attacks that involve some sorting of edges in *x* and y.

4.6.4. Line Segment Intersections. For each of the following segment pairs, determine whether the segments intersect, and if so where.

1. A = (1, 4), *B* = (7, 1/2), *C* = (7/2, 5/2), *D* = (7, 5);
2. A = (1, 4), *B* = (7, 1/2), *C* = (5, 0), *D* = (0, 7);
3. *A* = (0, 7), *B* = (7, 0), *C* = (8, - 1), *D* = (10, - 3);

4.6.1. Application of Line Intersections: the circle through three points.

Suppose a designer wants a tool that draws the unique circle that passes through three given points. The user specifies three points *A*, *B*, and *C*, on the display with the mouse as suggested in Figure 4.36a, and the circle is drawn automatically as shown in Figure 4.36b. The unique circle that passes through three points is called the **excircle** or **circumscribed circle**, of the triangle defined by the points. Which circle is it? We need a routine that can calculate its center and radius.

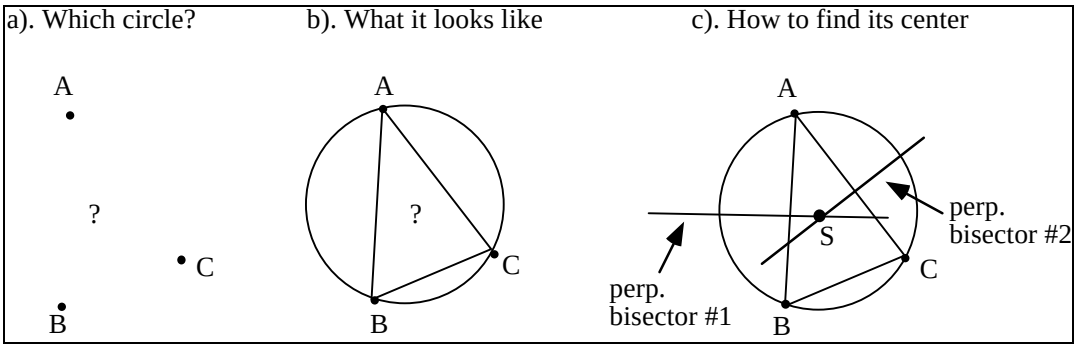


Figure 4.36. Finding the excircle.

Figure 4.35c shows how to find it. The center *S* of the desired circle must be equidistant from all three vertices, so it must lie on the **perpendicular bisector** of *each* side of triangle *ABC* (The perpendicular bisector is the locus of all points that are equidistant from two given points.). Thus we can determine *S* if we can compute where two of the perpendicular bisectors intersect.

We first show how to find a parametric representation of the perpendicular bisector of a line segment. Figure 4.37 shows a segment *S* with endpoints *A* and *B*. Its perpendicular bisector *L* is the infinite line that passes through the midpoint *M* of segment *S*, and is oriented perpendicular to it. But we know that midpoint *M* is given by $(A + B)/2$, and the direction of the normal is given by $(B-A)^{\perp}$, so the perpendicular bisector has parametric form:

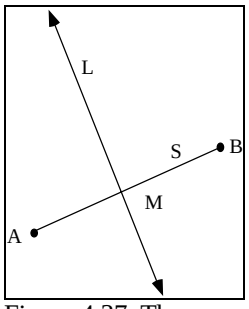


Figure 4.37. The perpendicular bisector of a segment.

$$L(t) = \frac{1}{2}(A+B) + (B-A)^{\perp} t \qquad \text{(the perpendicular bisector of } AB\text{)} \qquad (4.54)$$

Now we are in a position to compute the excircle of three points. Returning to Figure 4.35b we seek the intersection S of the perpendicular bisectors of AB and AC. For convenience we define the vectors:

$$\begin{aligned} \mathbf{a} &= B - A \\ \mathbf{b} &= C - B \\ \mathbf{c} &= A - C \end{aligned} \qquad (4.55)$$

To find the perpendicular bisector of AB we need the midpoint of AB and a direction perpendicular to AB. The midpoint of AB is $A + \mathbf{a} / 2$ (why?). The direction perpendicular to AB is $\mathbf{a}^{\perp}$. So the parametric form for the perpendicular bisector is $A + \mathbf{a} / 2 + \mathbf{a}^{\perp} t$. Similarly the perpendicular bisector of AC is $A - \mathbf{c} / 2 + \mathbf{c}^{\perp} u$, using parameter u. Point S lies where these meet, at the solution of:

$$\mathbf{a}^{\perp} t = \mathbf{b} / 2 + \mathbf{c}^{\perp} u$$

(where we have used $\mathbf{a} + \mathbf{b} + \mathbf{c} = \mathbf{0}$). To eliminate the term in u take the dot product of both sides with $\mathbf{c}$, and obtain $t = 1/2\ (\mathbf{b} \cdot \mathbf{c}) / (\mathbf{a}^{\perp} \cdot \mathbf{c})$. To find S use this value for t in the representation of the perpendicular bisector: $A + \mathbf{a} / 2 + \mathbf{a}^{\perp} t$, which yields the simple *explicit* form[10]:

$$S = A + \frac{1}{2}\left(\mathbf{a} + \frac{\mathbf{b} \cdot \mathbf{c}}{\mathbf{a}^{\perp} \cdot \mathbf{c}}\,\mathbf{a}^{\perp}\right) \qquad \text{(center of the excircle)} \qquad (4.56)$$

The radius of the excircle is the distance from S to any of the three vertices, so it is $|S - A|$. Just form the magnitude of the last term in Equation 4.56. After some manipulation (check this out) we obtain:

$$radius = \frac{|\mathbf{a}|}{2}\sqrt{\left(\frac{\mathbf{b} \cdot \mathbf{c}}{\mathbf{a}^{\perp} \cdot \mathbf{c}}\right)^2 + 1} \qquad \text{(radius of the excircle)} \qquad (4.57)$$

Once S and the radius are known, we can use `drawCircle()` from Chapter 3 to draw the desired circle.

Example 4.6.2. Find the perpendicular bisector L of the segment S having endpoints $A = (3, 5)$ and $B = (9, 3)$.
Solution: By direct calculation, midpoint $M = (6, 4)$, and $(B-A)^{\perp} = (2, 6)$, so L has representation $L(t) = (6 + 2t, 4 + 6t)$. It is useful to plot both S and L to see this result.

[10] Other closed form expressions for S have appeared previously, e.g. in [goldman90] and [lopex92]

Every triangle also has an **inscribed circle,** which is sometimes necessary to compute in a computer-aided design context. A case study examines how to do this, and also discusses the beguiling **nine-point circle.**

Practice Exercise 4.6.5. A Perpendicular Bisector. Find a parametric expression for the perpendicular bisector of the segment with endpoints A = (0, 6) and B = (4, 0). Plot the segment and the line.

4.7. Intersections of Lines with Planes, and Clipping.

The task of finding the intersection of a line with another line or with a plane arises in a surprising variety of situations in graphics. We have already seen one approach in Section 4.6, that finds where two line segments intersect. That approach used parametric representations for both the line segments, and solved two simultaneous equations.

Here we develop an alternative method that works for both lines and planes. It represents the intersecting line by a parametric representation, and the line or plane being intersected in a point normal form. It is very direct and clearly reveals what is going on. We develop the method once, and then apply the results to the problem of clipping a line against a convex polygon in 2D, or a convex polyhedron in 3D. In Chapter 7 we see that this is an essential step in viewing 3D objects. In Chapter 14 we use the same intersection technique to get started in ray tracing.

In 2D we want to find where a line intersects another line; in 3D we want to find where a line intersects a plane. Both of these problems can be solved at once because the formulation is in terms of dot products, and the same expressions arise whether the involved vectors are 2D or 3D. (We also address the problem of finding the intersection of two planes in the exercises: it too is based on dot products.)

Consider a line described parametrically as $R(t) = A + \mathbf{c}\, t$. We also refer to it as a "ray". We want to compute where it intersects the object characterized by the point normal form $\mathbf{n} \cdot (P - B) = 0$. In 2D this is a line; in 3D it is a plane. Point B lies on it, and vector $\mathbf{n}$ is normal to it. Figure 4.38a shows the ray hitting a line, and part b) shows it hitting a plane. We want to find the location of the "hit point".

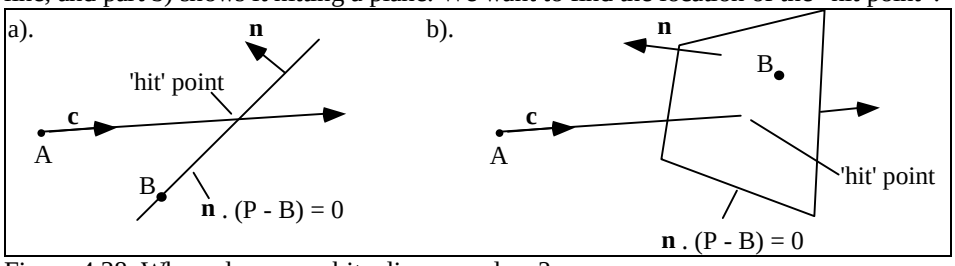


Figure 4.38. Where does a ray hit a line or a plane?

Suppose it hits at $t = t_{\text{hit}}$, the "hit time". At this value of t the line and ray must have the same coordinates, so $A + \mathbf{c}\, t_{\text{hit}}$ must satisfy the equation of the point normal form of the line or plane. Therefore we substitute this unknown "hit point" into the point normal equation to obtain a condition on t_{hit}:

$\mathbf{n} \cdot (A + \mathbf{c}\, t_{\text{hit}} - B) = 0.$

This may be rewritten as

$\mathbf{n} \cdot (A - B) + \mathbf{n} \cdot \mathbf{c}\, t_{hit} = 0,$

which is a linear equation in t_{hit}. Its solution is:

$$t_{hit} = \frac{\mathbf{n} \cdot (B - A)}{\mathbf{n} \cdot \mathbf{c}} \qquad \text{(hit time — 2D and 3D cases)} \qquad (4.58)$$

As always with a ratio of terms we must examine the eventuality that the denominator of t_{hit} is zero. This occurs when $\mathbf{n}\cdot\mathbf{c} = 0$, or when the ray is aimed parallel to the plane, in which case there is no hit at all.[11]

When the hit time has been computed, it is simple to find the location of the hit point : Substitute t_{hit} into the representation of the ray:

"hit" point: $P_{hit} = A + \mathbf{c}t_{hit}$ (hit spot — 2D and 3D cases) (4.59)

In the intersection problems treated below we will also need to know generally which direction the ray strikes the line or plane: "along with" the normal **n** or "counter to" **n**. (This will be important because we will need to know whether the ray is exiting from an object or entering it.) Figure 4.39 shows the two possibilities for a ray hitting a line. In part a) the angle between the ray's direction, **c**, and **n** is less than 90^0 so we say the ray is aimed "along with" **n**. In part b) the angle is greater than 90^o so the ray is aimed "counter to" **n**.

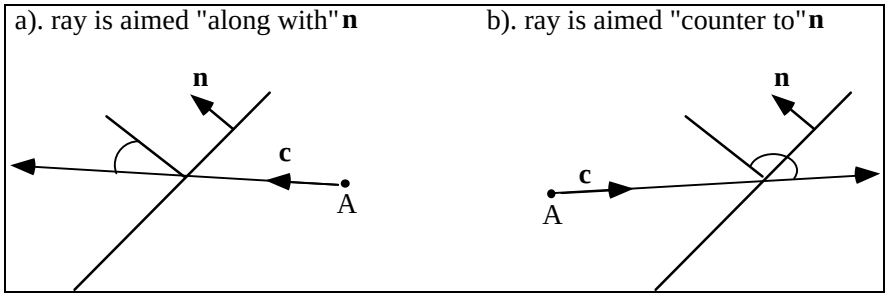


Figure 4.39. The direction of the ray is "along" or "against" **n**.

It is easy to test which of these possibilities occurs, since the *sign* of $\mathbf{n}\cdot\mathbf{c}$ tells immediately whether the angle between n and c is less than or greater than 90^o. Putting these ideas together, we have the three possibilities:

- if $\mathbf{n}\cdot\mathbf{c} > 0$ the ray is aimed "along with" the normal;
- if $\mathbf{n}\cdot\mathbf{c} = 0$ the ray is parallel to the line (4.60)
- if $\mathbf{n}\cdot\mathbf{c} < 0$ the ray is aimed "counter to" the normal

Practice Exercises.

4.7.1. Intersections of rays with lines and planes. Find when and where the ray $A + \mathbf{c}t$ hits the object $\mathbf{n}\cdot(P - B) = 0$ (lines in the 2D or planes in the 3D).
a). $A = (2, 3)$, $\mathbf{c} = (4, -4)$, $\mathbf{n} = (6,8)$, $B = (7,7)$.
b). $A = (2, -4, 3)$, $\mathbf{c} = (4, 0, -4)$, $\mathbf{n} = (6,9, 9)$, $B = (-7, 2, 7)$.
c). $A = (2, 0)$, $\mathbf{c} = (0 -4)$, $\mathbf{n} = (0,8)$, $B = (7,0)$.
d). $A = (2, 4, 3)$, $\mathbf{c} = (4, 4, -4)$, $\mathbf{n} = (6,4, 8)$, $B = (7, 4, 7)$.

4.7.2. Rays hitting Planes. Find the point where the ray $(1,5,2) + (5, -2, 6)t$ hits the plane $2x - 4y + z = 8$.

4.7.3. What is the intersection of two planes? Geometrically we know that two planes intersect in a straight line. But which line? Suppose the two planes are given by $\mathbf{n}\cdot(P - A) = 0$ and $\mathbf{m}\cdot(P - B) = 0$. Find the parametric form of the line in which they intersect. You may find it easiest to:
a). First obtain a parametric form for one of the planes: say, $C + \mathbf{a}s + \mathbf{b}t$ for the second plane.
b). Then substitute this form into the point normal form for the first plane, thereby obtaining a linear equation that relates parameters s and t.
c). Solve for s in terms of t, say $s = E + Ft$. (Find expressions for E and F.)
d). Write the desired line as $C + \mathbf{a}(E + Ft) + \mathbf{b}t$.

4.8. Polygon Intersection Problems.

[11] If the numerator is also 0 the ray lies entirely in the line (2D) or plane (3D). (why?).

We know polygons are the fundamental objects used in both 2D and 3D graphics. In 2D graphics their straight edges make it easy to describe them and draw them. In 3D graphics, an object is often modeled as a polygonal "mesh": a collection of polygons that fit together to make up its "skin". If the skin forms a closed surface that encloses some space the mesh is called a polyhedron. We study meshes and polyhedra in depth in Chapter 6.

Figure 4.40 shows a 2D polygon and a 3D polyhedron that we might need to analyze or render in a graphics application. Three important questions that arise are:

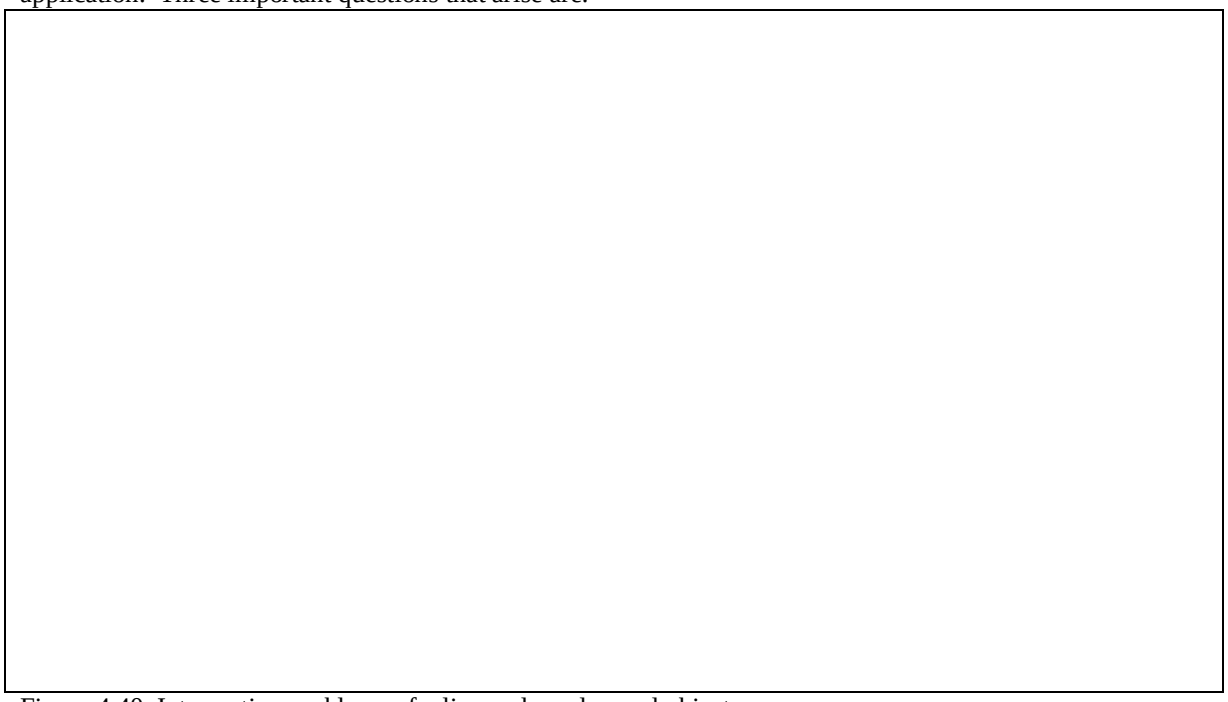


Figure 4.40. Intersection problems of a line and a polygonal object.

a). Is a given point *P* inside or outside the object?
b). Where does a given ray *R* first intersect the object?
c). Which part of a given line *L* lies inside the object, and which part lies outside?

As a simple example, which part(s) of the line $3y - 2x = 6$ lie inside the polygon whose vertices are (0, 3), (-2, -2), (-5, 0), (0, -7), (1, 1)?

4.8.1. Working with convex polygons and polyhedra.

The general case of intersecting a line with any polygon or polyhedron is quite complex; we address it in Section 4.8.4. Things are much simpler when the polygon or polyhedron is convex. They are simpler because a convex polygon is completely described by a set of "bounding lines"; in 3D a convex polyhedron is completely described by a set of "bounding planes". So we need only test the line against a set of unbounded lines or planes.

Figure 4.41 illustrates this for the 2D case. Part a shows a convex pentagon, and part b) shows the bounding lines L_0, L_1, etc. of the pentagon. Each bounding line defines two half spaces: the inside half space that contains the polygon, and the outside half space that shares no points with the polygon. Part c) of the figure shows a portion of the outside half space associated with the bounding line L_2.

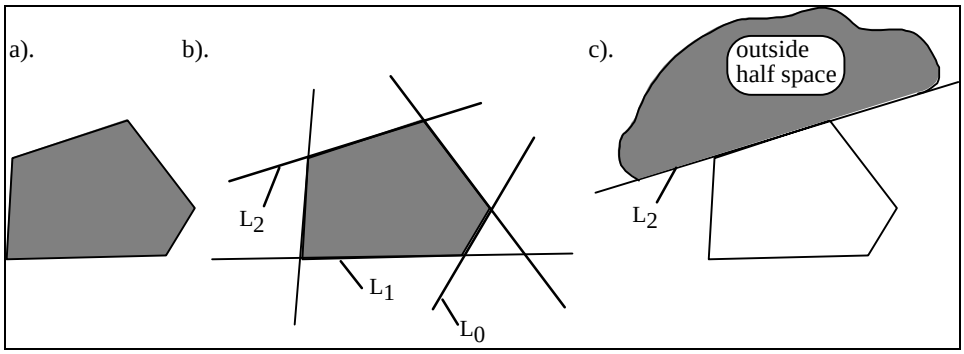


Figure 4.41. Convex polygons and polyhedra.

Example 4.8.1. Finding the bounding lines. Figure 4.42a shows a unit square. There are four bounding lines, given by $x = 1$, $x = -1$, $y = 1$, and $y = -1$. In addition, for each bounding line we can identify the outward normal vector: the one that points into the outside half space of the bounding line. The outward normal vector for the line $y = 1$ is of course $\mathbf{n} = (0, 1)$. (What are the other three?)

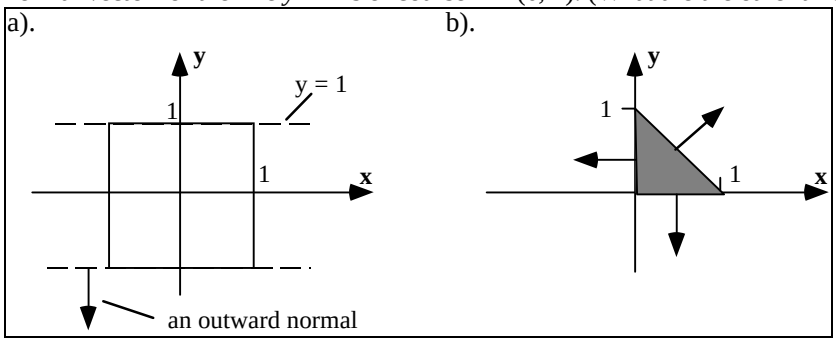


Figure 4.42. Examples of convex polygons.

The triangle in part b) has three bounding lines. (What is the equation for each line?) The point normal form for each of the three lines is given next; in each case it uses the outward normal (check this):

$(-1, 0) \cdot (P - (0, 0)) = 0;$
$(0, -1) \cdot (P - (0, 0)) = 0;$
$(1, 1) \cdot (P - (1, 0)) = 0;$

The big advantage in dealing with convex polygons is that we perform intersection tests only on infinite lines, and don't need to check whether an intersection lies "beyond" an endpoint — recall the complexity of the intersection tests in Section 4.7. In addition the point normal form can be used, which simplifies the calculations.

For a convex polyhedron in 3D, each plane has an inside and an outside half space, and an outward pointing normal vector. The polyhedron is the intersection of all the inside half spaces, (the set of all points that are simultaneously in the inside half space of every bounding plane).

4.8.2. Ray Intersections and Clipping for Convex Polygons.

We developed a method in Section 4.7 that finds where a ray hits an individual line or plane. We can use this method to find where a ray hits a convex polygon or polyhedron.

The Intersection Problem. Where does the ray $A + \mathbf{c}t$ hit polygon P?

Figure 4.43 shows a ray $A + \mathbf{c}\,t$ intersecting polygon P. We want to know all of the places where the ray hits P. Because P is convex the ray hits P exactly twice: It enters once and exits once. Call the values of t at which it enters and exits t_{in} and t_{out}, respectively. The ray intersection problem is to compute the values of t_{in} and t_{out}. Once these hit times are known we of course know the hit points themselves:

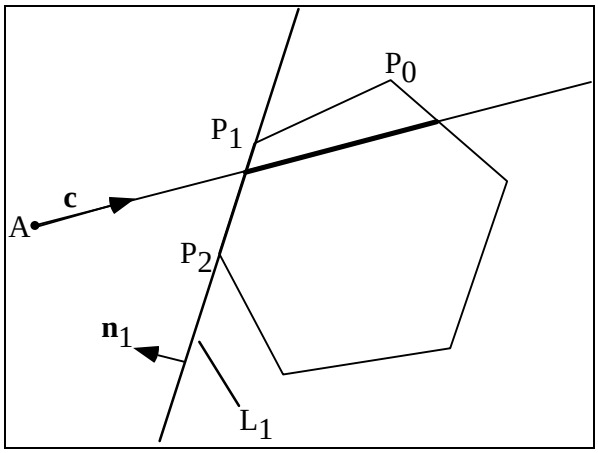


Figure 4.43. Ray $A + \mathbf{c}t$ intersecting a convex polygon.

Entering hit point: $A + \mathbf{c}\, t_{in}$ (4.61)
Exiting hit point: $A + \mathbf{c}\, t_{out}$

The ray is inside P for all t in the interval $[t_{in}, t_{out}]$.

Note that finding t_{in} and t_{out} not only solves the intersection problem, but also the clipping problem. If we know t_{in} and t_{out} we know which part of the line $A + \mathbf{c}t$ lies inside P. Usually the clipping problem is stated as:

The Clipping problem: For the two points A and C which part of segment AC lies inside P?

Figure 4.44 shows several possible situations. Part a) shows the case where A and C both lie outside P, but there is a portion of the segment AC that lies inside P.

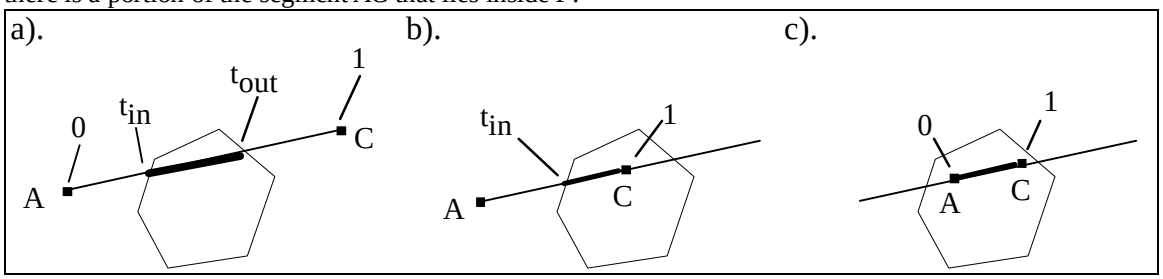


Figure 4.44. A segment clipped by a polygon.

If we consider segment AC as part of a ray given by $A + \mathbf{c}t$ where $\mathbf{c} = C - A$, then point A corresponds to the point on the ray at $t = 0$, and C corresponds to the point at $t = 1$. These "ray times" are labeled in the figure. To find the clipped segment we compute t_{in} and t_{out} as described above. The segment that "survives" clipping has end points $A + \mathbf{c}\, t_{in}$ and $A + \mathbf{c}\, t_{out}$. In Figure 4.43b point C lies inside P and so t_{out} is larger than 1. The clipped segment has end points $A + \mathbf{c}\, t_{in}$ and C. In part c) both A and C lie inside P, so the clipped segment is the same: AC.

In general we compute t_{in}, and compare it to 0. The larger of the values 0 and t_{in} is used as the "time" for the first end point of the clipped segment. Similarly, the smaller of the values 1 and t_{out} is used to find the second end point. So the end points of the clipped segment are:

$A' = A + \mathbf{c}\ max(0, t_{in})$ (4.62)
$C' = A + \mathbf{c}\ min(t_{out}, 1)$

Now how are t_{in} and t_{out} computed? We must consider each of the bounding lines of P in turn, and find where the ray $A + \mathbf{c}\, t$ intersects it. We suppose each bounding line is stored in point normal form as the pair

{*B*, **n**}, where *B* is some point on the line and **n** is the *outward pointing normal* for the line: it points to the outside of the polygon. Because it is outward pointing the test of Equation 4.60 translates to:

if $\mathbf{n} \cdot \mathbf{c} > 0$	the ray is exiting from *P*;	
if $\mathbf{n} \cdot \mathbf{c} = 0$	the ray is parallel to the line	(4.63)
if $\mathbf{n} \cdot \mathbf{c} < 0$	the ray is entering *P*	

For each bounding line we find:
a). The hit time of the ray with the bounding line (use Equation 4.58);
b). Whether the ray is entering or exiting the polygon (use Equation 4.63)

If the ray is entering, we know that the time at which the ray ultimately enters *P* (if it enters it at all) cannot be *earlier* than this newly found hit time. We keep track of the "earliest possible entering time as t_{in}. For each entering hit time, t_{hit}, we replace t_{in} by $\max(t_{in}, t_{hit})$. Similarly we keep track of the *latest* possible exit time as t_{out}, and for each exiting hit we replace t_{out} by $\min(t_{out}, t_{hit})$.

It helps to think of the interval $[t_{in}, t_{out}]$ as the **candidate interval** of *t*, the interval of *t* inside of which the ray *might* lie inside the object. Figure 4.45 shows an example for the clipping problem. We know the point $A + \mathbf{c}t$ *cannot* be inside *P* for any *t* in the candidate interval. As each bounding line is tested, the candidate interval gets reduced as t_{in} is increased or t_{out} is decreased: pieces of it get "chopped" off. To get started we initialize t_{in} to 0 and t_{out} to 1 for the line clipping problem, so the candidate interval is [0,1].

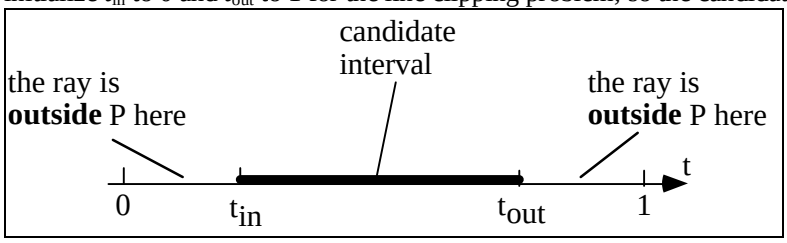


Figure 4.45. The candidate interval for a hit.

The algorithm is then:
1). Initialize the candidate interval to [0,1][12].
2). For each bounding line, use Equation 4.58 to find the hit time t_{hit} and determine whether it's an entering or exiting hit:
- if it's an entering hit, set $t_{in} = \max(t_{in}, t_{hit})$
- if it's an exiting hit, set $t_{out} = \min(t_{out}\ t_{hit})$

If at any point t_{in} becomes greater than t_{out} we know the ray misses *P* entirely, and testing is terminated
3). If candidate interval is not empty, then from Equation 4.62 the segment from $A + \mathbf{c}\ t_{in}$ to $A + \mathbf{c}\ t_{out}$ is known to lie inside *P*. For the line clipping problem these are the endpoints of the clipped line. For the ray intersection problem we know the entering and exiting points of the ray.

Note that we stop further testing as soon the candidate interval vanishes. This is called an **early out:** if we determine early in the processing that the ray is outside of the polygon, we save time by immediately exiting from the test.

Figure 4.46 shows a specific example of clipping: we seek the portion of segment *AC* that lies in polygon *P*. We initialize t_{in} to 0 and t_{out} to 1. The ray "starts" at *A* at $t = 0$ and proceeds to point *C*, reaching it at $t = 1$. We test it against each bounding line L_0, L_1, .., in turn and update t_{in} and t_{out} as necessary.

[12] For the ray intersection problem, where the ray extends infinitely far in both directions, we set $t_{in} = -\infty$ and $t_{out} = \infty$. In practice t_{in} is set to a large negative value, and t_{out} to a large positive value.

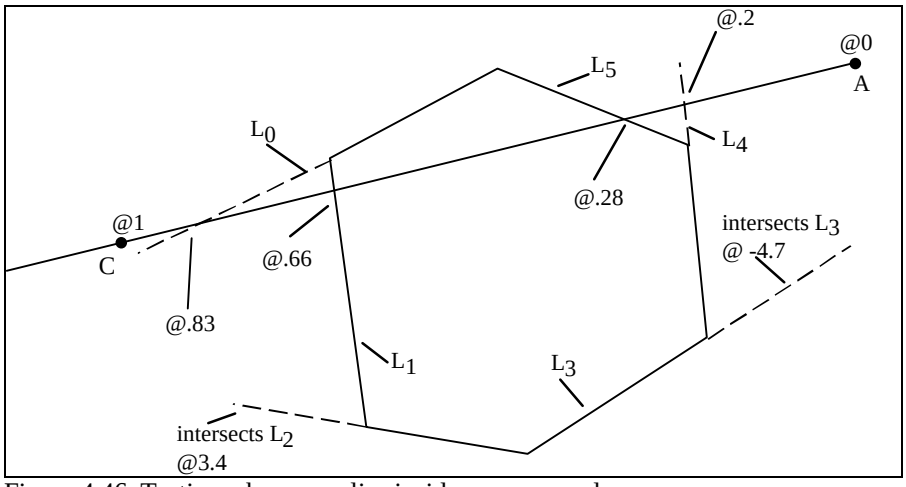


Figure 4.46. Testing when a ray lies inside a convex polygon.

Suppose when we test it against line L_0 we find an exiting hit at $t = 0$ .83. This sets t_{out} to 0.83, and the candidate interval is now [0, 0.83]. We then test it against L_1 and find an exiting hit at $t = 0.66$. This reduces the candidate interval to [0, 0.66]. The test against L_2 gives an exiting hit at $t = 3.4$. This tells us nothing new: we already know the ray is outside for $t > 0.66$. The test against L_3 gives an entering hit at $t = -0.47$. So we set t_{in} to -0.47, and the candidate interval is [-0.47, 0.66]. The test with L_4 gives an entering hit at $t = 0.2$, so t_{in} is updated to 0.2. Finally, testing against L_5 gives an entering hit at $t = 0.28$, and we are done. The candidate interval is [0.28, 0.66].In fact the ray *is* inside P for all t between 0.28 and 0.66.

Figure 4.47 shows the sequence of updates to t_{in} and t_{out} that occur as each of the lines above is tested.

line test	t_{in}	t_{out}
0	0	0.83
1	0	0.66
2	0	0.66
3	0	0.66
4	0.2	0.66
5	0.28	0.66

Figure 4.47. Updates on the values of t_{in} and t_{out}.

4.8.3. The Cyrus-Beck Clipping Algorithm.

We build a routine from these ideas, that performs the clipping of a line segment against any convex polygon. The method was originally developed by Cyrus and Beck [cyrus78]. Later a highly efficient clipper for rectangular windows was devised by Liang and Barsky [liang84] based on similar ideas. It is discussed in a Case Study at the end of this chapter.

The routine that implements the Cyrus-Beck clipper has interface:

```
int CyrusBeckClip(Line& seg, LineList& L);
```

Its parameters are the line segment, `seg`, to be clipped (which contains the first and second endpoints named `seg.first` and `seg.second`) and the list of bounding lines of the polygon. It clips `seg` against each line in `L` as described above, and places the clipped segment back in `seg.` (This is why `seg` must be passed by reference.) The routine returns:

- 0 if no part of the segment lies in *P* (the candidate interval became empty);
- 1 if some part of the segment does lie in *P*.

Figure 4.48 shows pseudocode for the Cyrus Beck algorithm. The types `LineSegment`, `LineList`, and `Vector2` are suitable data types to hold the quantities in question (see the exercises). Variables `numer` and `denom` hold the numerator and denominator for t_{hit} of Equation 4.48:

$$numer = \mathbf{n} \cdot (B - A)$$
$$denom = \mathbf{n} \cdot \mathbf{c} \qquad (4.64)$$

```
int CyrusBeckClip(LineSegment& seg, LineList L)
{
 double numer, denom; // used to find hit time for each line
 double tIn = 0.0, tOut = 1.0;
 Vector2 c, tmp;
  form vector: c = seg.second - seg.first
 for(int i = 0; i < L.num; i++) // chop at each bounding line
 {
    form vector tmp = L.line[i].pt - first
    numer = dot(L.line[i].norm, tmp);
    denom = dot(L.line[i].norm, c);
    if(!chopCI(numer, denom, tIn, tOut)) return 0; // early out
 }
  // adjust the endpoints of the segment; do second one 1st.
 if (tOut < 1.0 ) // second endpoint was altered
 {
     seg.second.x  = seg.first.x + c.x * tOut;
     seg.second.y  = seg.first.y + c.y * tOut;
 }
 if (tIn > 0.0)  // first endpoint was altered
 {
     seg.first.x  = seg.first.x + c.x * tIn;
     seg.first.y  = seg.first.y + c.y * tIn;
}
      return 1; // some segment survives
}
```

Figure 4.48. Cyrus-Beck Clipper for a Convex Polygon, 2D case (pseudocode).

Note that the value of `seg.second` is updated first, since we must use the old value of `seg.first` in the update calculation for both `seg.first` and `seg.second.`

The routine `chopCI()` is shown in Figure 4.49. It uses `numer` and `denom` of Equation 4.64 to calculate the hit time at which the ray hits a bounding line, uses Equation 4.63 to determine whether the ray is entering or exiting the polygon, and "chops" off the piece of the candidate interval CI that is thereby found to be outside the polygon.

```
int chopCI(double& tIn, double& tOut, double numer, double
denom)
{
  double tHit;
  if (denom < 0)           // ray is entering
  {
     tHit = numer / denom;
     if (tHit > tOut) return 0;  // early out
     else if (tHit > tIn) tIn = tHit; // take larger t
  }
  else if(denom > 0)          // ray is exiting
  {
     tHit = numer / denom;
     if(tHit < tIn) return 0;      // early out
     if(tHit < tout) tOut = tHit; // take smaller t
  }
  else             // denom is 0: ray is parallel
  if(numer <= 0) return 0;   // missed the line

  return 1; // CI is still non-empty
}
```

Figure 4.49. Clipping against a single bounding line.

If the ray is parallel to the line it could lie entirely in the inside half space of the line, or entirely out of it. It turns out that *numer* = **n** · (*B* - *A*) is exactly the quantity needed to tell which of these cases occurs. See the exercises.

The 3D case: Clipping a line against a Convex Polyhedron.
The Cyrus Beck clipping algorithm works in three dimensions in exactly the same way. In 3D the edges of the window become planes defining a convex region in three dimensions, and the line segment is a line suspended in space. `ChopCI`() needs no changes at all (since it uses only the values of dot products - through `numer` and `denom`). The data types in `CyrusBeckClip`() must of course be extended to 3D types, and when the endpoints of the line are adjusted the z-component must be adjusted as well.

Practice Exercises.
4.8.2. Data types for variable in the Cyrus Beck Clipper. Provide useful definitions for data types, either as struct's or classes, for `LineSegment`, `LineList`, and `Vector2` used in the Cyrus Beck clipping algorithm.
4.8.3. What does numer <= 0 do?
Sketch the vectors involved in value of `numer` in `chopCI()` and show that when the ray *A* + **c** *t* moves parallel to the bounding line **n** · (*P* - *B*) = 0, it lies wholly in the inside half space of the line if and only if `numer` > 0.
4.8.4. Find the Clipped Line. Find the portion of the segment with endpoints (2, 4) and (20, 8) that lies within the quadrilateral window with corners at (0, 7), (9, 9), (14,4), and (2, 2).

4.8.4. Clipping against arbitrary polygons.

We saw how to clip a line segment against a convex polygon in the previous section. We generalize this to a method for clipping a segment against *any* polygon.

The basic problem is to find where the ray *A* + **c***t* lies inside polygon *P* given by the vertex list P_0, P_1, .., P_{N-1}. Figure 4.50 shows an example.

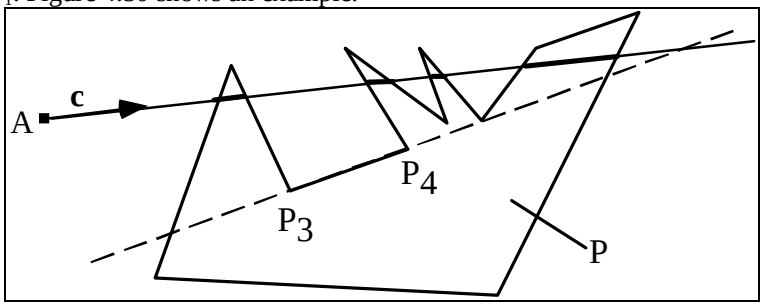


Figure 4.50. Where is a ray inside an arbitrary polygon *P*?

It is clear that the ray can enter and exit from *P* multiple times in general, and that the result of clipping a segment against P may result in a *list* of segments rather than a single one. Also, of course, *P* is no longer described by a collection of infinite bounding lines in point normal form; we must work with the *N* finite segments such as P_3 P_4 that form its edges.

The problem is close to the problem we dealt with in Section 4.7: finding the intersection of two line segments. Now we are intersecting one line segment with the sequence of line segments associated with *P*.

We represent each edge of *P* parametrically (rather than in point normal form). For instance, the edge P_3 P_4 is represented as P_3 + $\mathbf{e}_3 u$ where $\mathbf{e}_3 = P_4 - P_3$ is the **edge vector** associated with P_3. In general, the *i*-th edge is given by P_i + $\mathbf{e}_i u$, for *u* in [0,1] and *i* = 0, 1, ..., *N*-1where $\mathbf{e}_i = P_{i+1} - P_i$, and as always we equate P_N with P_0.

Recall from Section 4.7 that the ray *A* + **c***t* hits the *i*-th edge when *t* and *u* have the proper values to make *A* + **c***t* = P_i + **e***i* *u*. Calling vector $\mathbf{b}_i = P_i - A$ we seek the solution (values of *t* and *u*) of

$\mathbf{c}t = \mathbf{b}_i + \mathbf{e}_i u$

Equations 4.51 and 4.52 hold the answers. When converted to the current notation we have:

$$t = \frac{\mathbf{e}_i^{\perp} \cdot \mathbf{b}_i}{\mathbf{e}_i^{\perp} \cdot \mathbf{c}} \qquad \text{and} \qquad u = \frac{\mathbf{c}^{\perp} \cdot \mathbf{b}_i}{\mathbf{e}_i^{\perp} \cdot \mathbf{c}}.$$

If $\mathbf{e}_i^{\perp} \cdot \mathbf{c}$ is 0 the *i*-th edge is parallel to the ray direction **c** and there is no intersection. There is a true intersection with the *i*-th edge only if *u* falls in the interval [0,1].

We need to find all of the legitimate hits of the ray with edges of *P*, and place them in a list of the hit times. Call this list `hitList`. Then pseudocode for the process would look like:

```
initialize hitList to empty
for(int i = 0; i < N; i++)    // for each edge of P
{
      build bi, ei for the i-th edge
      solve for t, u
      if(u lies in [0,1])
            add t to the hitList
}
```

What we do now with this list depends on the problem at hand.

The ray intersection problem. (Where does the ray *first* hit *P*?)
This is solved by finding the smallest value of *t*, t_{min}, in `theList`. The hit spot is, as always, $A + \mathbf{c}\, t_{min}$.

The line clipping problem.
For this we need the sequence of *t*-intervals in which the ray is inside *P*. This requires sorting `theList` and then taking the *t*-values in pairs. The ray enters *P* at the first time in each pair, and exits from *P* at the second time of each pair.

Example 4.9.2. Clip *AB* to polygon *P*. Suppose the line to be clipped is *AB* as shown in Figure 4.51, for which *A* = (1, 1) and *B* = (8, 2).

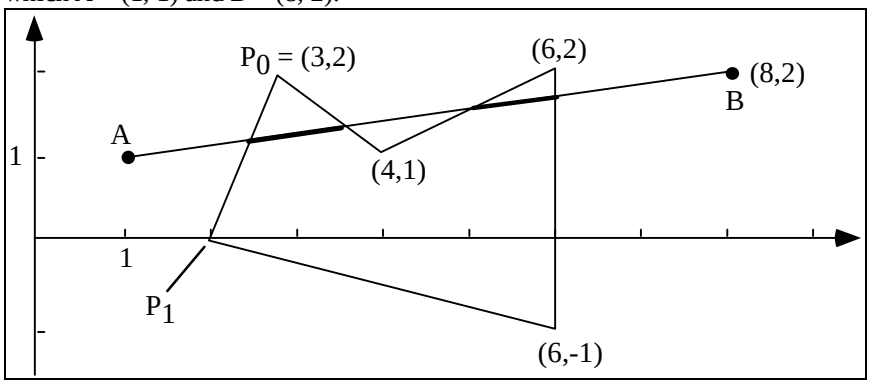


Figure 4.51. Clipping a line against a polygon.

P is given by the vertex list: (3, 2), (2, 0), (6, -1), (6, 2), (4, 1). Taking each edge in turn we get for the values of *t* and *u* at the intersections:

edge	*u*	*t*
0	0.3846	0.2308
1	-0.727	-0.2727
2	0.9048	0.7142
3	0.4	0.6

4 0.375 0.375

The hit with edge 1 occurs at t outside of [0,1] so it is discarded. We sort the remaining t-values and arrive at the sorted hit list: {0.2308, 0.375, 0.6, 0.7142}. Thus the ray enters P at t = 0.2308, exits it at t = 0.375, re-enters it at t = 0.6, and exits it for the last time at t = 0.7142 .

Practice exercise 4.9.4. Clip a line. Find the portions of the line from A = (1, 3.5) to B = (9, 3.5) that lie inside the polygon with vertex list: (2, 4), (3, 1), (4, 4), (3, 3).

4.8.5. More Advanced Clipping.

Clipping algorithms are fundamental to computer graphics, and a number of efficient algorithms have been developed. We have examined two approaches to clipping so far. The **Cohen Sutherland** clipping algorithm, studied in Chapter 2, clips a line against an aligned rectangle. The **Cyrus-Beck** clipper generalizes this to clipping a line against any convex polygon or polyhedron. But situations arise where one needs more sophisticated clipping. We mention two such methods here, and develop details of both in Case Studies at the end of this chapter.

The **Sutherland–Hodgman** clipper is similar to the Cyrus–Beck method, performing clipping against a convex polygon. But instead of clipping a single line segment, it clips an entire polygon (which needn't be convex) against the convex polygon. Most importantly, its output is again a *polygon* (or possibly a set of polygons). It can be important to retain the polygon structure during clipping since the clipped polygons may need to be filled with a pattern or color. This is not possible if the edges of the polygon are clipped individually.

The **Weiler–Atherton** clipping algorithm clips any polygon, *P*, against *any* other polygon, *W*, convex or not. It can output the part of *P* that lies inside *W* (**interior clipping**) or the part of *P* that lies outside W (**exterior clipping**). In addition, both *P* and *W* can have "holes" in them. As might be expected, this algorithm is somewhat more complex than the others we have examined, but its power makes it a welcome addition to one's toolbox in a variety of applications.

4.9. Summary of the Chapter.

Vectors provide a convenient way to express many geometric relations, and the operations they support provide a powerful way to manipulate geometric objects algebraically. Many computer graphics algorithms are simplified and made more efficient through the use of vectors. Because most vector operations are expressed the same way independent of the dimensionality of the underlying space, it is possible to derive results that are equally true in 2D or 3D space.

The dot product of two vectors is a fundamental quantity that simplifies finding the length of a vector and the angle between two vectors. It can be used to find such things as the orthogonal projection of one vector onto another, the location of the center of the excircle of three points, and the direction of a reflected ray. It is often used to test whether two vectors are orthogonal to one another, and more generally to test when they are pointing less than, or more than, 90^0 from each other. It is also useful to work with a 2D vector $\mathbf{a}^{\perp}$ that lies 90^0 to the left of a given vector **a.** In particular the dot product $\mathbf{a}^{\perp} \cdot \mathbf{b}$ reports useful information about how **a** and **b** are disposed relative to each other.

The cross product also reveals information about the angle between two vectors in 3D, and in addition evaluates to a vector that is perpendicular to them both. It is often used to find a vector that is normal to a plane.

In the process of developing an algorithm it is crucial to have a concise representation of the graphical objects involved. The two principal forms are the parametric representation, and the implicit form. The parametric representation "visits" each of the points on the object as a parameter is made to vary, so the parameter "indexes into" different points on the object. The implicit form expresses an equation that all points on the object, and only those, must satisfy. It is often given in the form $f(x, y) = 0$ in 2D, or $f(x, y, z) = 0$ in 3D, where $f()$ is some function. The value of $f()$ for a given point not only tells when the point is on

the object, but when a point lies off of the object the sign of *f*() can reveal on *which* side of the object the point lies. In this chapter we addressed finding representations of the two fundamental "flat" objects in graphics: lines and planes. For such objects both the parametric form and implicit form are linear in their arguments. The implicit form can be revealingly written as the dot product of a normal vector and a vector lying within the object.

It is possible to form arbitrary linear combinations of vectors, but not of points. For points only affine combinations are allowed, or else chaos reigns if the underlying coordinate system is ever altered, as it frequently is in graphics. Affine combinations of points are useful in graphics, and we showed that they form the basis of "tweening" for animations and for Bezier curves.

The parametric form of a line or ray is particularly useful for such tasks as finding where two lines intersect or where a ray hits a polygon or polyhedron. These problems are important in themselves, and they also underlie clipping algorithms that are so prominent in graphics. The Cyrus-Beck clipper, which finds where a line expressed parametrically shares the same point in space as a line or plane expressed implicitly, addresses a larger class of problems than the Cohen Sutherland clipper of Chapter 2, and will be seen in action in several contexts later.

In the Case Studies that are presented next, the vector tools developed so far are applied to some interesting graphics situations, and their power is seen even more clearly. Whether or not you intend to carry out the required programming to implement these mini-projects, it is valuable to read through them and imagine what process you would pursue to solve them.

4.10. Case Studies.

4.10.1. Case Study 4.1: Animation with Tweening.

(Level of Effort: II.)) Devise two interesting polylines, such as *A* and *B* as shown in Figure 4.52. Ensure that *A* and *B* have the same number of points, perhaps by adding an artificial extra point in the top segment of *B*.

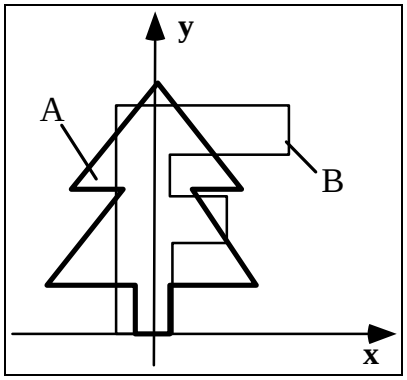


Figure 4.52. Tweening two polylines.

a). Develop a routine similar to routine `drawTween(A, B, n, t)`of Figure 4.23 that draws the tween at *t* of the polylines *A* and *B*.

b). Develop a routine that draws a sequence of "tweens" between *A* and *B* as *t* varies from 0 to 1, and experiment with it. Use the double buffering offered by OpenGL to make the animation smooth.

c). Extend `the routine so` that after *t* increases gradually from 0 to 1 it decreases gradually back to 0 and then repeats, so the animation repeatedly shows *A* mutating into *B* then back into *A*. This should continue until a key is pressed.

d). Arrange so that the user can enter two polylines with the mouse, following which the polylines are tweened as just described. The user presses key 'A' and begins to lay down points to form polyline *A*, then presses key 'B' and lays down the points for polyline B. Pressing 'T' terminates that process and begins the tweening, which continues until the user types 'Q'. Allow for the case where the user inputs a different number of points for *A* than for *B*: your program automatically creates the required number of extra points along line segments (perhaps at their midpoints) of the polyline having fewer points.

4.10.2. Case Study 4.2. Circles Galore.

(Level of Effort: II.). Write an application that allows the user to input the points of a triangle with a mouse. The program then draws the triangle along with its **inscribed circle, excircle**, and **9-point circle**, each in a different color. Arrange matters so the user can then move vertices of the tirangle to new locations with the mouse, whereupon the new triangle with its three circles are redrawn.

We saw how to draw the excircle in Section 4.6.1. Here we show how to find the inscribed circle and the nine-point circle.

T**he inscribed circle**. This is the circle that just snugs up inside the given triangle, and is tangent to all three sides[13]. Figure 4.53a shows a triangle *ABC* along with its inscribed circle.

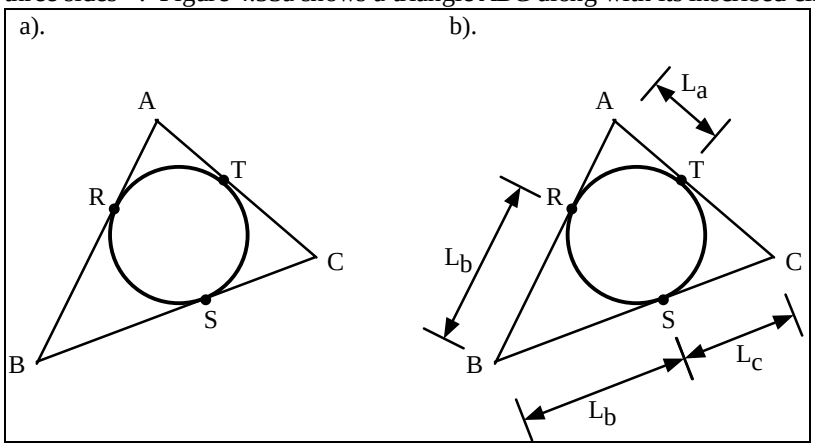


Figure 4.53. The inscribed circle of *ABC* is the excircle of *RST*.

As was the case with the excircle, the hard part is finding the center of the inscribed circle. A straightforward method[14] recognizes that the inscribed circle of *ABC* is simply the excircle of a different set of three points, *RST* as shown in Figure 4.53a.

We need only find the locations of *R*, *S*, and *T* and then use the excircle method of Section 4.6.1. Figure 4.53b shows the distances of *R*, *S*, and *T* from *A*, *B*, and *C*. By the symmetry of a circle the distances $|B - R|$ and $|B - S|$ must be equal, and there are two other pairs of lines that have the same length. We therefore have (using the definitions of Equation 4.55 for **a**, **b**, and **c**):

$$|\mathbf{a}| = L_b + L_a, \qquad |\mathbf{b}| = L_b + L_c, \qquad |\mathbf{c}| = L_a + L_c$$

which can be combined to solve for L_a and L_b:

$$2\,L_a = |\mathbf{a}| + |\mathbf{c}| - |\mathbf{b}|, \qquad 2\,L_b = |\mathbf{a}| + |\mathbf{b}| - |\mathbf{c}|$$

so L_a and L_b are now known. Thus *R*, *S*, and *T* are given by:

[13]Note: finding the incircle also solves the problem of finding the unique circle that is tangent to 3 noncollinear lines in the plane.

[14] Suggested by Russell Swan.

$$R = A + L_a \frac{\mathbf{a}}{|\mathbf{a}|}$$
$$S = B + L_b \frac{\mathbf{b}}{|\mathbf{b}|} \qquad (4.65)$$
$$T = A - L_a \frac{\mathbf{c}}{|\mathbf{c}|}$$

(Check these expressions!)

Encapsulate the calculation of *R*, *S*, and *T* from *A*, *B*, and *C* in a simple routine having usage `getTangentPoints(A, B, C, R, S, T)`. The advantage here is that if we have a routine `excircle()` that takes three points and computes the center and radius of the excircle defined by them, we can use the *same* routine to find the inscribed circle. Experiment with these tools.

The nine-point circle.
For any triangle, there are nine particularly distinguished points:
• the midpoints of the 3 sides;
• the feet of the 3 altitudes;
• the midpoints of the lines joining the orthocenter (where the 3 altitudes meet) to the vertices.

Remarkably, a single circle passes through all nine points! Figure 4.54 shows **the 9-point circle**[15] for an example triangle. The nine-point circle is perhaps most easily drawn as the excircle of the midpoints of the sides of the triangle.

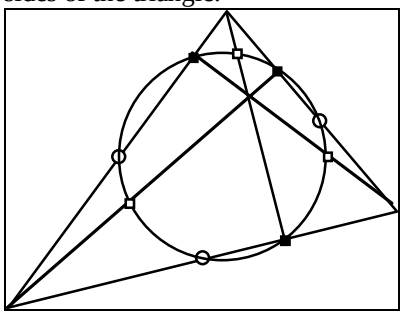


Figure 4.54. The 9-point circle.

4.10.3. Case Study 4.3. Is point *Q* inside convex polygon *P*?

(Level of Effort: II.) Suppose you are given the specification of a convex polygon, *P*. Then given a point *Q* you are asked to determine whether or not *Q* lies inside *P*. But from the discussion on convex polygons in Section 4.8.1 we know this is equivalent to asking whether *Q* lies on the inside half space of *every* bounding line of *P*. For each bounding line L_i we need only test whether the vector $Q - P_i$ is more than 90^0 away from the outward pointing normal.

Fact: *Q* lies in *P* if $(Q - P_i) \cdot n_i < 0$ for $i = 0, 1, ..., N-1$. (4.66)

Figure 4.55 illustrates the test for the particular bounding line that passes through P_1 and P_2. For the case of point *Q*, which lies inside *P*, the angle with $\mathbf{n}_1$ is greater than 90^0. For the case of point *Q*' which lies outside *P* the angle is less than 90^0.

[15] "This circle is the first really exciting one to appear in any couse on elementary geometry." Daniel Pedoe. *Circles*, Pergamon Press, New York, 1957

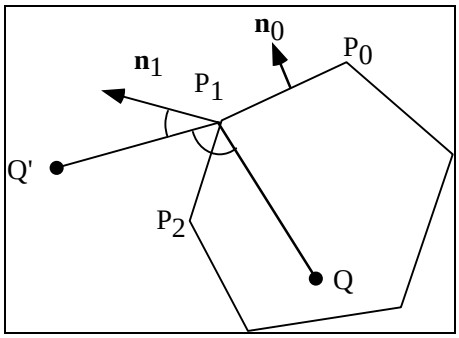


Figure 4.55. Is point *Q* inside polygon *P*?

Write and test a program that allows the user to:
a). lay down the vertices of a convex polygon, *P*, with the mouse;
b). successively lay down test points, Q, with the mouse;
c). prints "is inside" or "is not inside" depending on whether the point *Q* is or is not inside *P*.

4.10.4. Case Study 4.4. Reflections in a Chamber (2D Ray Tracing)

(Level of Effort: II.) This case study applies some of the tools and ideas introduced in this chapter to a fascinating yet simple simulation. The simulation performs a kind of ray tracing, based in a 2D world for easy visualization. Three dimensional ray tracing is discussed in detail in Chapter 14.

This simulation traces the path of a single tiny "pinball" as it bounces off various walls inside a "chamber." Figure 4.56a shows a cross section of a convex chamber *W* that has six walls and contains three convex "pillars". The pinball begins at point *S* and moves in a straight line in direction **c** until it hits a barrier, whereupon it "reflects" off the barrier and moves in a new direction, again in a straight line. It continues to do this forever. Figure 4.56b shows an example of the polyline path that a ray traverses.

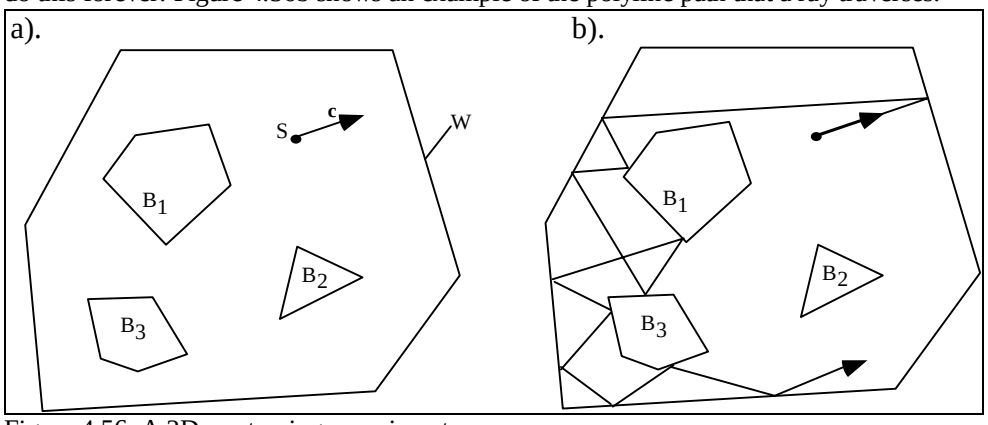


Figure 4.56. A 2D ray-tracing experiment.

For any given position *S* and direction **c** of the ray, tracing its path requires two operations:

• Finding the first wall of the chamber "hit" by the ray;
• Finding the new direction the ray will take as it reflects off this first line.

Both of these operations have been discussed in the chapter. Note that as each new ray is created, its start point is always on some wall, the "hit point" of the previously hit wall.

We represent the chamber by a list of convex polygons, $pillar_0$, $pillar_1$, ..., and arrange that $pillar_0$ is the "chamber" inside which the action takes place. The pillars are stored in suitable arrays of points. For each ray beginning at *S* and moving in direction **c**, the entire array of pillars is scanned, and the intersection of the ray with each pillar is determined. This test is done using the Cyrus-Beck algorithm of Section 4.8.3. If

there is a hit with a pillar, the "hit time" is taken to be the time at which the ray "enters" the pillar. We encapsulate this test in the routine:

```
int rayHit(Ray thisRay, int which, double& tHit);
```

that calculates the hit time `tHit` of the ray `thisRay` against $pillar_{which}$ and returns 1 if the ray hits the pillar, and 0 if it misses. A suitable type for `Ray` is `struct{Point2 startPt; Vector2 dir;}` or the corresponding class; it captures the starting point *S* and direction **c** of the ray.

We want to know which pillar the ray hits first. This is done by keeping track of the earliest hit time as we scan through the list of pillars. Only positive hit times need to be considered: negative hit times correspond to hits at spots in the opposite direction from the ray's travel. When the earliest hit point is found, the ray is drawn from *S* to it.

We must find the direction of the reflected ray as it moves away from this latest hit spot. The direction **c**' of the reflected ray is given in terms of the direction **c** of the incident ray by Equation 4.27:

$$\mathbf{c}' = \mathbf{c} - 2(\mathbf{c} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} \qquad (4.67)$$

where $\hat{\mathbf{n}}$ is the unit normal to the wall of the pillar that was hit. If a pillar inside the chamber was hit we use the outward pointing normal; if the chamber itself was hit, we use the inward pointing normal.

Write and exercise a program that draws the path of a ray as it reflects off the inner walls of chamber *W* and the walls of the convex pillars inside the chamber. Arrange to read in the list of pillars from an external file and to have the user specify the ray's starting position and direction. (Also see Chapter 7 for the "elliptipool" 2D ray tracing simulation.)

4.10.5. Case Study 4.5. Cyrus-Beck Clipping.

(Level of Effort: II.) Write and exercise a program that clips a collection of lines against a convex polygon. The user specifies the polygon by laying down a sequence of points with the mouse (pressing key 'C' to terminate the polygon and begin clipping). Then a sequence of lines is generated, each having randomly chosen end points.
For each such line, the whole line is first drawn in red, then the portion that lies inside the polygon is drawn in blue.

4.10.6. Case Study 4.6. Clipping a polygon against a convex polygon — Sutherland Hodgman Clipping.

(Level of Effort: III.) Clipping algorithms studied so far clip individual line segments against polygons. When instead a *polygon* is clipped against a window it can be fragmented into several polygons in the clipping process, as suggested in Figure 4.57a. The polygon may need to be filled with a color or pattern, which means that each of the clipped fragments must be associated with that pattern, as suggested in Figure 4.57b. Therefore a clipping algorithm must keep track of edges *ab*, *cd*, and so on, and must fashion a new polygon (or polygons) out of the original one. It is also important that an algorithm not retain extraneous edges such as *bc* as part of the new polygon, as such edges would be displayed when they should in fact be invisible.

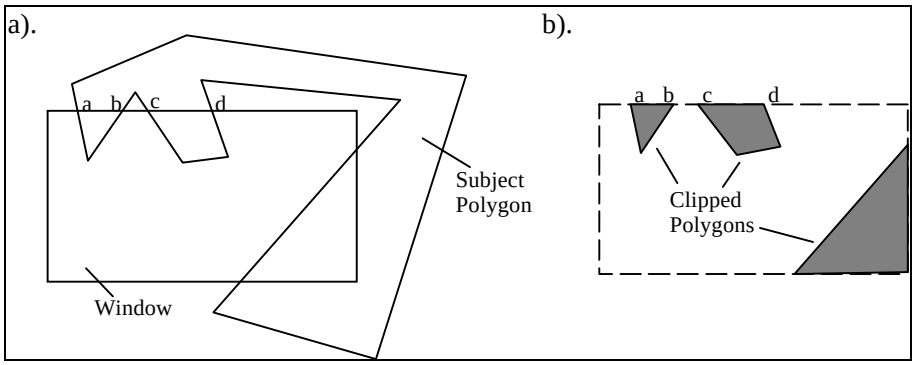


Figure 4.57. Clipping a polygon against a polygon.

The polygon to be clipped will be called the "subject" polygon, *S*. The polygon against which *S* is clipped will be called the "clip" polygon, *C*. How do we clip polygon *S*, represented by a vertex list, against polygon *C*, to generate a collection of vertex lists that properly represent the set of clipped polygons?

We examine here the **Sutherland–Hodgman** clipping algorithm**.** This method is quite simple and clips any subject polygon (convex or not) against a convex clip polygon. The algorithm can leave extraneous edges that must be removed later.

Because of the many different cases that can arise, we need an organized method for keeping track of the clipping process. The Sutherland–Hodgman algorithm takes a divide-and-conquer approach: It breaks a difficult problem into a set of simpler ones. It is built on the Cyrus-Beck approach, but must work with a *list* of vertices - that represent a polygon - rather than a simple pair of vertices.

Like the Cyrus-Beck algorithm this method clips polygon *S* against each bounding line of polygon *C* in turn, leaving only the part that is inside *C*. Once all of the edges of *C* have been used this way, *S* will have been clipped against *C* as desired. Figure 4.58 shows the algorithm in action for

1st edition Figure A6.2 on page 716.

Figure 4.58. Sutherland–Hodgman polygon clipping.

a seven-sided subject polygon *S* and a rectangular clip polygon *C*. We will describe each step in the process for this example. *S* is characterized by the vertex list *a b c d e f g*. *S* is clipped against the top, right, bottom, and left edges of *C* in turn, and at each stage a new list of vertices is generated from the old. This list describes one or more polygons and is passed along as the subject polygon for clipping against the next edge of *C*.

The basic operation, then, is to clip the polygon(s) described by an input vertex list *V* against the current clip edge of *C* and produce an output vertex list. To do this, traverse *V*, forming successive edges with pairs of adjacent vertices. Each such edge *E* has a first and a second endpoint we call *s* and *p*, respectively. There are four possible situations for endpoints *s* and *p*: *s* and *p* can both be inside, both can be outside, or they can be on opposite sides of the clip edge. In each case, certain points are output to (appended onto) the new vertex list, as shown in Figure 4.59.

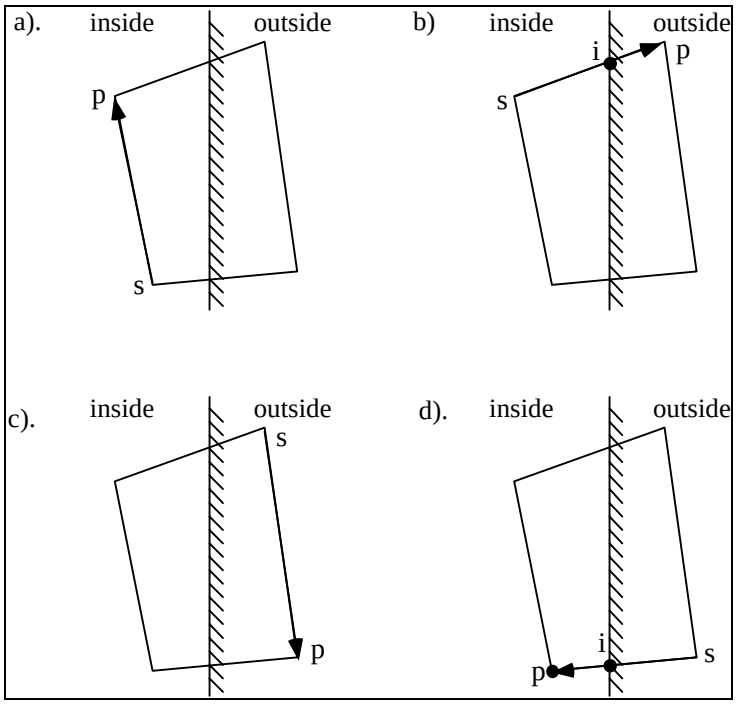


Figure 4.59. Four cases for each edge of S.

a. Both *s* and *p* are inside: *p* is output.
b. *s* is inside and *p* is outside. Find the intersection *i* and output it.
c. Both *s* and *p* are outside. Nothing is output.
d. *s* is outside and *p* is inside. Find intersection *i*, and output *i* and then *p*.

Now follow the progress of the Sutherland–Hodgman algorithm in Figure 4.58. Consider clipping *S* against the top edge of *C*. The input vertex list for this phase is *a b c d e f g*. The first edge from the list is taken for convenience as that from *g* to *a*, the edge that "wraps around" from the end of the list to its first element. Thus point *s* is *g* and point *p* is *a* here. Edge *g*, *a*, meaning the edge from *g* to *a*, intersects the clip edge at a new point "*1*", which is output to the new list. (The output list from each stage in the algorithm is shown below the subsequent figure in Figure 4.58.) The next edge in the input list is *a*, *b*. Since both endpoints are above the clipping edge, nothing is output. The third edge, *b*, *c*, generates two output points, 2 and *c*, and the fourth edge, *c*, *d*, outputs point *d*. This process continues until the last edge, *f*, *g*, is tested, producing *g*. The new vertex list for the next clipping stage is therefore 1 2 *c d e f g*. It is illuminating to follow the example in Figure 4.58 carefully in its entirety to see how the algorithm works.

Notice that extraneous edges 3, 6 and 9, 10 are formed that connect the three polygon fragments formed in the clipping algorithm. Such edges can cause problems in some polygon filling algorithms. It is possible but not trivial to remove these offending edges [sutherland74].

Task: Implement the Sutherland-Hodgman clipping algorithm, and test it on a variety of sample polygons. The user lays down the convex polygon *C* with the mouse, then lays down the subject polygon *S* with the mouse. It is drawn in red as it is being laid down. Clipping is then performed, and the clipped polygon(s) are drawn in blue.

4.10.7. Case Study 4.7. Clipping a Polygon against another — Weiler Atherton Clipping.

(Level of Effort: III). This method provides the most general clipping mechanism of all we have studied. It clips any subject polygon against any (possibly non-convex) clip polygon. The polygons may even contain holes.

The Sutherland-Hodgman algorithm examined in Case Study 4.6 exploits the convexity of the clipping polygon through the use of inside-outside half-spaces. In some applications, such as hidden surface removal and rendering shadows, however, one must clip one concave polygon against another. Clipping is more complex in such cases. The Weiler–Atherton approach clips any polygon against any other, even when they have holes. It also allows one to form the set theoretic **union**, **intersection**, and **difference** of two polygons, as we discuss in Case Study 4.8.

We start with a simple example, shown in Figure 4.60. Here two concave polygons, *SUBJ* and *CLIP*, are represented by the vertex lists, (*a*, *b*, *c*, *d*) and (*A*, *B*, *C*, *D*), respectively. We adopt the convention here of listing vertices so that the interior of the polygon is to the right of each edge as we move cyclically from vertex to vertex through the list. For instance, the interior of *SUBJ* lies to the right of the edge from *c* to *d* and to the right of that from *d* to *a*. This is akin to listing vertices in "clockwise" order.

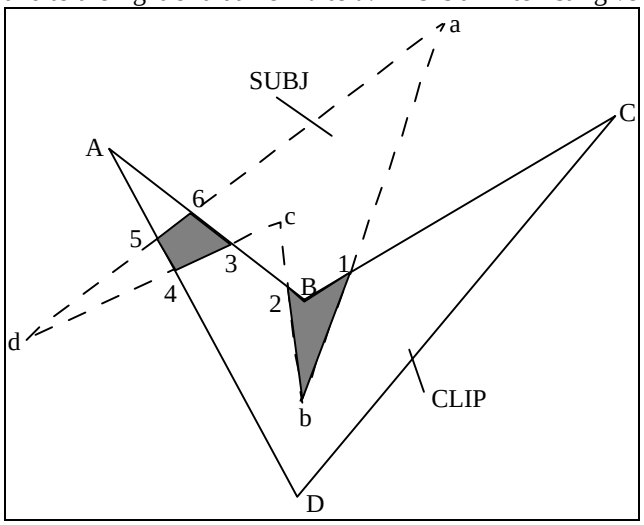


Figure 4.60 .Weiler–Atherton clipping.

All of the intersections of the two polygons are identified and stored in a list (see later). For the example here, there are six such intersections. Now to clip *SUBJ* against *CLIP*, traverse around *SUBJ* in the "forward direction" (i.e., so that its interior is to the right) until an "entering" intersection is found: one for which *SUBJ* is moving from the outside to the inside of *CLIP*. Here we first find 1, and it goes to an output list that records the clipped polygon(s).

The process is now simple to state in geometric terms: Traverse along *SUBJ*, moving segment by segment, until an intersection is encountered (2 in the example). The idea now is to turn away from following *SUBJ* and to follow *CLIP* instead. There are two ways to turn. Turn so that *CLIP* is traversed in its forward direction. This keeps the inside of both *SUBJ* and *CLIP* to the right. Upon finding an intersection, turn and follow along *SUBJ* in its forward direction, and so on. Each vertex or intersection encountered is put on the output list. Repeat the "turn and jump between polygons" process, traversing each polygon in its forward direction, until the first vertex is revisited. The output list at this point consists of (1, *b*, 2, *B*).

Now check for any other entering intersections of *SUBJ*. Number 3 is found and the process repeats, generating output list (3, 4, 5, 6). Further checks for entering intersections show that they have all been visited, so the clipping process terminates, yielding the two polygons (1, *b*, 2, *B*) and (3, 4, 5, 6). An organized way to implement this "follow in the forward direction and jump" process is to build the two lists

SUBJLIST: *a*, 1, *b*, 2, *c*, 3, 4, *d*, 5, 6
CLIPLIST: *A*, 6, 3, 2, *B*, 1, *C*, *D*, 4, 5

that traverse each polygon (so that its interior is to the right) and list both vertices and intersections in the order they are encountered. (What should be done if no intersections are detected between the two

polygons?) Therefore traversing a polygon amounts to traversing a list, and jumping between polygons is effected by jumping between lists.

Notice that once the lists are available, there is very little geometry in the process—just a "point outside polygon" test to properly identify an entering vertex. The proper direction in which to traverse each polygon is embedded in the ordering of its list. For the preceding example, the progress of the algorithm is traced in Figure 4.61.

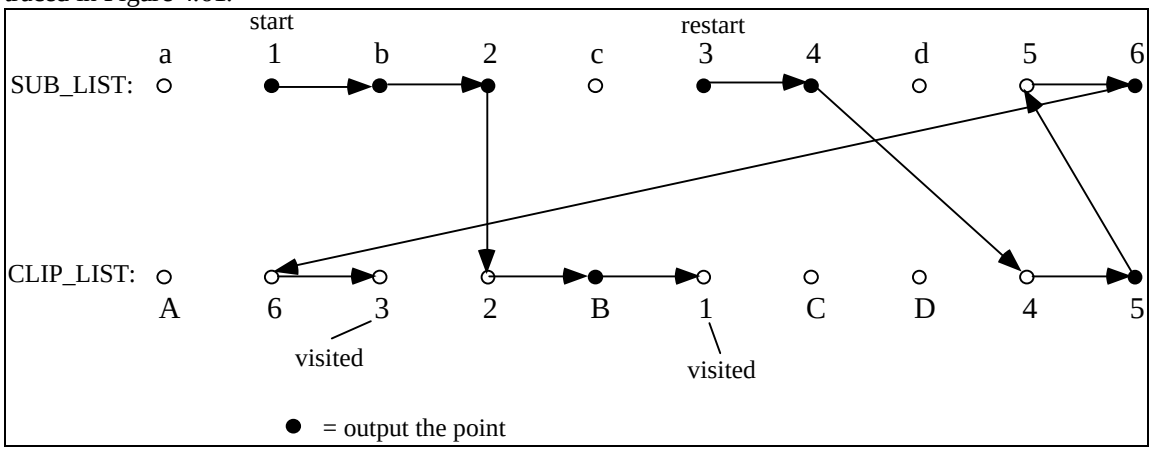


Figure 4.61. Applying the Weiler–Atherton method.

A more complex example involving polygons with holes is shown in Figure 4.62. The

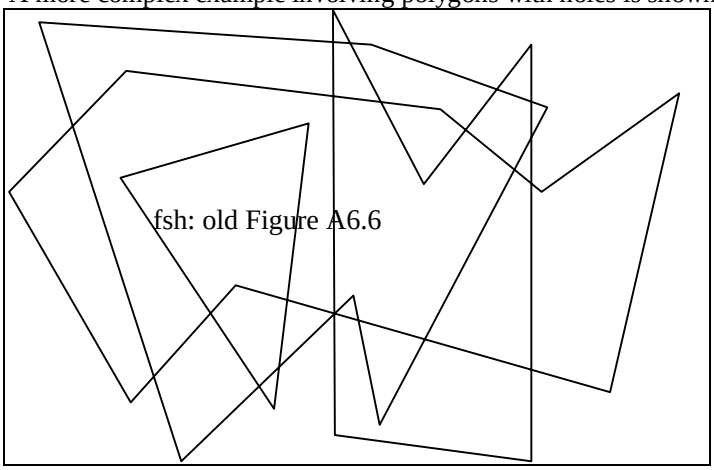


Figure 4.62. Weiler–Atherton clipping: polygons with holes.

vertices that describe holes are also listed in order such that the interior of the polygon lies to the right of an edge. (For holes this is sometimes called "counterclockwise order.") The same rule is used as earlier: Turn and follow the other polygon in its forward direction. Beginning with entering intersection *1*, the polygon (1, 2, 3, 4, 5, *i*, 6, *H*) is formed. Then, starting with entering intersection 7, the polygon (7, 8, 9, *c*, 10, *F*) is created. What entering intersection should be used to generate the third polygon? It is a valuable exercise to build *SUBJLIST* and *CLIPLIST* and to trace through the operation of the method for this example.

As with many algorithms that base decisions on intersections, we must examine the preceding method for cases where edges of *CLIP* and *SUBJ* are parallel and overlap over a finite segment.

Task: Implement the Weiler-Atherton clipping algorithm, and test it on a variety of polygons. Generate *SUBJ* and *CLIP* polygons, either in files or by letting the user lay down polygons with the mouse. In your implementation carefully consider how the algorithm will operate in situations such as the following:
• Some edges of *SUBJ* and *CLIP* are parallel and overlap over a finite segment,
• *SUBJ* or *CLIP* or both are nonsimple polygons,

• Some edges of *SUBJ* and *CLIP* overlap only at their endpoints,
• *CLIP* and *SUBJ* are disjoint,
• *SUBJ* lies entirely within a hole of *CLIP*.

4.10.8. Case Study 4.8. Boolean Operations on Polygons.

(Level of Effort: III.) If we view polygons as sets of points (the set of all points on the boundary or in the interior of the polygon), then the result of the previous clipping operation is the **intersection** of the two polygons, the set of all points that are in both *CLIP* and *SUBJ*. The polygons output by the algorithm consist of points that lie both within the original *SUBJ* and within the *CLIP* polygons. Here we generalize from intersections to other set theoretic operations on polygons, often called "Boolean" operations. Such operations arise frequently in modeling [mortenson85] as well as in graphics (see Chapter 14). In general, for any two sets of points A and B, the three set theoretic operations are

• intersection: $A \cap B$ = {all points in both *A* and *B*}
• union: $A \cup B$ = {all points in *A* or in *B* or both}
• difference: *A* - *B*= {all points in *A* but not *B*}

with a similar definition for the set difference *B* - *A*. Examples of these sets are shown in Figure 4.63.

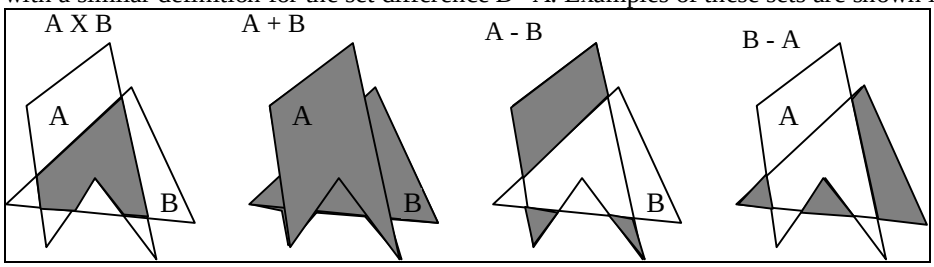


Figure 4.63. Polygons formed by boolean operations on polygons.

It is not hard to adjust the Weiler-Atherton method, which already performs intersections, to perform the union and difference operations on polygons *A* and *B*.

1. Computing the union of A and B. Traverse around *A* in the forward direction until an exiting intersection is found: one for which *A* is moving from the inside to the outside of *B*. Output the intersection and traverse along *A* until another intersection with *B* is found. Now turn to follow *B* in its forward direction. At each subsequent intersection, output the vertex and turn to follow the other polygon in its forward direction. Upon returning to the initial vertex, look for other exiting intersections that have not yet been visited.

2. Computing the difference A - B(outside clipping). Whereas finding the intersection of two polygons results in clipping one against the other, the difference operation "shields" one polygon from another. That is, the difference *SUBJ - CLIP* consists of the parts of *SUBJ* that lie outside *CLIP*. No parts of *SUBJ* are drawn that lie within the border of *CLIP*, so the region defined by *CLIP* is effectively protected, or shielded.

Traverse around *A* until an entering intersection into *B* is found. Turn to *B*, following it in the reverse direction, (so that *B*'s interior is to the left). Upon reaching another intersection, jump to *A* again. At each intersection, jump to the other polygon, always traversing *A* in the forward direction and *B* in the reverse direction. Some examples of forming the union and difference of two polygons are shown in Figure 4.64. The three set operations generate the following polygons:

POLYA ∪ POLYB:
4, 5, *g*, *h*(a hole)
8, *B*, *C*, *D*, 1, *b*, *c*, *d*
2, 3, *i*, *j*(a hole)
6, *H*, *E*, *F*, 7, *f* (a hole)

POLYA - POLYB:
4, 5, 6, *H*, *E*, *F*, 7, *e*, 8, *B*, *C*, *D*, *1*, a
2, 3, *k*

POLYB - POLYA:
1, *b*, *c*, *d*, 8, 5, *g*, *h*, 4, *A*, 3, *i*, *j*, 2
7, *f*, 6, *G*

1st edition Figure A6.8.

Figure 4.64. Forming the union and difference of two polygons.

Notice how the holes (*E*, *F*, *G*, *H*) and (*k*, *i*, *j*) in the polygons are properly handled, and that the algorithm generates holes as needed (holes are polygons listed in counterclockwise fashion).

Task: Adapt the Weiler–Atherton method so that it can form the union and difference of two polygons, and exercise your routines on a variety of polygons. Generate *A* and *B* polygons, either in files or algorithmically, to assist in the testing. Draw the polygons *A* and *B* in two different colors, and the result of the operation in a third color.

4.11. For Further Reading

Many books provide a good introduction to vectors. A favorite is Hoffmann's ABOUT VECTORS. The GRAPHICS GEMS series [gems] provides an excellent source of new approaches and results in vector arithmetic and geometric algorithms by computer graphics practitioners. Three excellent example articles are Alan Paeth's "A Half-Angle Identity for Digital Computation: The Joys of the Half Tangent" [paeth91], Ron Goldman "Triangles"[goldman90], and Lopez-Lopez's "Triangles Revisited" [lopez92]. Two books that delve more deeply into the nature of geometric algorithms are Moret and Shapiro's ALGORITHMS FROM P TO NP [moret91] and Preparata and Shamos's COMPUTATIONAL GEOMETRY, AN INTRODUCTION [preparata85].