Chapter 4: Mathematics for Modeling

The primary mathematical background needed for computer graphics programming is 3D analytic geometry. It is unusual to see a course with this title, however, so most students pick up bits and pieces of mathematics background that fill this in. One of the common sources of the background is introductory physics; another is multivariate calculus. Neither of these is a common requirement for computer graphics, however, so here we will outline the general concepts we will use in these notes.

Coordinate systems

The set of real numbers—often thought of as the set of all possible distances between points—is a mathematical abstraction that is effectively modeled as a Euclidean straight line with two uniquely-identified points. One point is identified with the number 0.0 (we will write all real numbers with decimals, to meet the expectations of programming languages), called the origin, and the other is identified with the number 1.0, which we call the unit point. The direction of the line from 0.0 to 1.0 is called the positive direction; the opposite direction of the line is called the negative direction. These directions identify the parts of the lines associated with positive and negative numbers, respectively.

If we have two straight lines that are perpendicular to each other and meet in a point, we can define that point to be the origin for both lines, and choose two points the same distance from the origin on each line as the unit points. A distance unit is defined to be used by each of the two lines, and the points at this distance from the intersection point are marked, one to the right of the intersection and one above it. This gives us the classical 2D coordinate system, often called the Cartesian coordinate system. The vectors from the intersection point to the right-hand point (respectively the point above the intersection) are called the X- and Y-direction vectors and are indicated by $\mathbf{i}$ and $\mathbf{j}$ respectively. Points in this system are represented by an ordered pair of real numbers, $(X, Y)$, and this is probably the most familiar coordinate system to most people. These points may also be represented by a vector $<X, Y>$ from the origin to the point, and this vector may be expressed in terms of the direction vectors as $Xi + Yj$.

In 2D Cartesian coordinates, any two lines that are not parallel will meet in a point. The lines make four angles when they meet, and the acute angle is called the angle between the lines. If two line segments begin at the same point, they make a single angle that is called the angle between the line segments. These angles are measured with the usual trigonometric functions, and we assume that the reader will have a modest familiarity with trigonometry. Some of the reasons for this assumption can be found in the discussions below on polar and spherical coordinates, and in the description of the dot product and cross product. We will discuss more about the trigonometric aspects of graphics when we get to that point in modeling or lighting.

The 3D world in which we will do most of our computer graphics work is based on 3D Cartesian coordinates that extend the ideas of 2D coordinates above. This is usually presented in terms of three lines that meet at a single point, which is identified as the origin for all three lines and is called the origin, that have their unit points the same distance from that point, and that are mutually perpendicular. Each point represented by an ordered triple of real numbers $(x, y, z)$. The three lines correspond to three unit direction vectors, each from the origin to the unit point of its respective line; these are named $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ for the X-, Y-, and Z-axis, respectively, and are called the canonical basis for the space, and the point can be represented as $xi + yj + zk$. Any ordered triple of real numbers is identified with the point in the space that lies an appropriate distance from the two-axis planes, with the first (x) coordinate being the distance from the Y-Z plane, the second (y) coordinate being the distance from the X-Z plane, and the third (z) coordinate being the distance from the X-Y plane. This is all illustrated in Figure 4.1.
3D coordinate systems can be either right-handed or left-handed: the third axis can be the cross product of the first two axes, or it can be the negative of that cross product, respectively. (We will talk about cross products a little later in this chapter.) The “handed-ness” comes from a simple technique: if you hold your hand in space with your fingers along the first axis and curl your fingers towards the second axis, your thumb will point in a direction perpendicular to the first two axes. If you do this with the right hand, the thumb points in the direction of the third axis in a right-handed system. If you do it with the left hand, the thumb points in the direction of the third axis in a left-handed system.

Some computer graphics systems use right-handed coordinates, and this is probably the most natural coordinate system for most uses. For example, this is the coordinate system that naturally fits electromagnetic theory, because the relationship between a moving current in a wire and the magnetic field it generates is a right-hand coordinate relationship. The modeling in Open GL is based on a right-hand coordinate system.

On the other hand, there are other places where a left-handed coordinate system is natural. If you consider a space with a standard X-Y plane as the front of the space and define Z as the distance back from that plane, then the values of Z naturally increase as you move back into the space. This is a left-hand relationship.

Points, lines, and line segments

In this model, any real number is identified with the unique point on the line that is
- at the distance from the origin which is that number times the distance from 0.0 to 1.0, and
- in the direction of the number’s sign.

We have heard that a line is determined by two points; let’s see how that can work. Let the first point be \( P_0 = (X_0, Y_0, Z_0) \) and the second point be \( P_1 = (X_1, Y_1, Z_1) \). Let’s call \( P_0 \) the origin and \( P_1 \) the unit point. Points on the segment are obtained by starting at the “first” point \( P_0 \) offset by a fraction of the difference vector \( P_1 - P_0 \). Then any point \( P = (X, Y, Z) \) on the line can be expressed in vector terms by

\[
P = P_0 + t(P_1 - P_0) = (1-t)P_0 + tP_1
\]

for a single value of a real variable \( t \). This computation is actually done for each coordinate, with a separate equation for each of X, Y, and Z as follows:

- \( X = X_0 + t(X_1 - X_0) = (1-t)X_0 + tX_1 \)
- \( Y = Y_0 + t(Y_1 - Y_0) = (1-t)Y_0 + tY_1 \)
- \( Z = Z_0 + t(Z_1 - Z_0) = (1-t)Z_0 + tZ_1 \)

Thus any line segment can be determined by a single parameter, which is why a line is called a one-dimensional object. Points along the line are determined by values of the parameter, as
illustrated in Figure 4.2 below that shows the coordinates of the points along a line segment determined by value of $t$ from 0 to 1 in increments of 0.25.

![Diagram](image)

Figure 4.2: a parametric line segment with points determined by some values of the parameter

This representation for a line segment (or an entire line, if you place no restrictions on the value of $t$) also allows you to compute intersections involving lines. The reverse concept is also useful, so if you have a known point on the line, you can calculate the value of the parameter $t$ that would produce that point. For example, if a line intersects a plane or another geometric object at a point $Q$, a vector calculation of the form $P_0 + t(P_1 - P_0) = Q$ would allow you to calculate the value of the parameter $t$ that gives the intersection point on the line. This calculation might involve only a single equation or all three equations, depending on the situation, but your goal is to compute the value of $t$ that represents the point in question. This is often the basis for geometric computations such as the intersection of a line and a plane.

As an example of this, we’ll take two points and calculate the parametric equations for the line. Let $P_0 = (3.0, 4.0, 5.0)$ and $P_1 = (5.0, -1.5, 4.0)$. Then $P_1 - P_0 = (2.0, -5.5, -1.0)$, so the equations of the line are

$$x = 3.0 + 2.0t$$
$$y = 4.0 - 5.5t$$
$$z = 5.0 - t$$

If we consider a plane $6.0x - 2.0y + 1.5z - 4.0 = 0.0$, the point where the line intersects the plane is given by $6.0(3.0 + 2.0t) - 2.0(4.0 - 5.5t) + 1.5(5.0 - t) - 4.0 = 0.0$. Combining terms yields $21.5t - 13.5 = 0$, or $t = 13.5/21.5 = 27/43$, from which the intersection point is $(183/43, -41/86, 188/43)$. Of course, you will rarely do this kind of calculation manually, but will code up such computations as needed in your program.

**Distance from a point to a line**

As an application of the concept of parametric lines, let’s see how we can compute the distance from a point in 3-space to a line. If the point is $P_0=(u,v,w)$ and the line is given by parametric equations

$$x = a + bt$$
$$y = c + dt$$
$$z = e + ft$$

then for any point $P=(x,y,z)$ on the line, the square of the distance from $P$ to $P_0$ is given by

$$(a + bt - u)^2 + (c + dt - v)^2 + (e + ft - w)^2$$

which is a quadratic equation in $t$. This quadratic is minimized by taking its derivative and looking for the point where the derivative is 0:

$$2b(a + bt - u) + 2d(c + dt - v) + 2f(e + ft - w) = 0$$

which is a simple linear equation in $t$, and its unique solution for $t$ allows you to calculate the point $P$ on the line which is nearest point $P_0$. 

12/30/01
Line segments and parametric curves

In standard Euclidean geometry, two points determine a line as we noted above. In fact, in the same way we talked about any line having unique origin identified with 0.0 and unit point identified with 1.0, a line segment—the points on the line between these two particular points—can be identified as the points corresponding to values between 0 and 1. It is done by much the same process as we used to illustrate the 1-dimensional nature of a line above. That is, just as in the discussion of lines above, if the two points are \( P_0 \) and \( P_1 \), we can identify any point between them as \( P = (1−t)P_0 + tP_1 \) for a unique value of \( t \) between 0 and 1. This is called the parametric form for a line segment.

The line segment gives us an example of determining a continuous set of points by functions from the interval \([0,1]\) to 3-space. In general, if we consider any set of continuous functions \( x(t), y(t), \) and \( z(t) \) that are defined on \([0,1]\), the set of points they generate is called a parametric curve in 3-space. There are some very useful applications of such curves. For example, you can display the locations of a moving point in space, you can compute the positions along a curve from which you will view a scene in a fly-through, or you can describe the behavior of a function of two variables on a domain that lies on a curve in 2-space.

Vectors

Vectors in 3-space are triples of real numbers written as \(<a, b, c>\). These may be identified with points, or they may be viewed as representing the motion needed to go from one point to another in space. The latter viewpoint will be one we use often.

The length of a vector is defined as the square root of the sum of the squares of the vector’s components, written \(|<a,b,c>|=\sqrt{a^2+b^2+c^2}\). A unit vector is a vector whose length is 1, and unit vectors are very important in a number of modeling and rendering computations; basically a unit vector can be treated as a pure direction. If \( V=<a,b,c> \) is any vector, we can make it a unit vector by dividing each of its components by its length: \( \frac{a}{\|V\|}, \frac{b}{\|V\|}, \frac{c}{\|V\|} \). Doing this is called normalizing the vector.

Dot and cross products of vectors

There are two computations on vectors that we will need to understand, and sometimes to perform, in developing the geometry for our graphic images. The first is the dot product of two vectors. This produces a single real value that represents the projection of one vector on the other and its value is the product of the lengths of the two vectors times the cosine of the angle between them. The dot product computation is quite simple: it is simply the sum of the componentwise products of the vectors. If the two vectors \( A \) and \( B \) are

\[
A = <X_1,Y_1,Z_1>
\]

\[
B = <X_2,Y_2,Z_2>
\]

then their dot product is computed as

\[
A \cdot B = X_1 \times X_2 + Y_1 \times Y_2 + Z_1 \times Z_2 .
\]

Note that a simple consequence of the definition is that the length of any vector \( A \) is the square root of the dot product \( A \cdot A \) of the vector with itself.
The observation about the cosine of the angle between the vectors is very important. If two vectors are parallel, the dot product is simply the product of their lengths, but if they are orthogonal—the angle between them is 90°—then the dot product is zero. If the angle between them is acute, then the dot product will be positive, no matter what the orientation of the vectors, because the cosine of any angle between –90° and 90° is positive; if the angle between them is obtuse, then the dot product will be negative. Note that another useful application of the fact is that you can compute the angle between two vectors if you know the vectors’ lengths and dot products. These observations about the dot product is very useful in a number of graphical computations.

The relationship between the dot product and the cosine of the included angle is very important. We can take advantage of it to look at the component of any vector that lies in the direction of another vector, which we call the projection of one vector on another. As we see in Figure 4.3, with any two vectors we can construct a right triangle in which one side is one of the vectors and the other is the projection of the first on the second. Because \( U \cdot V = ||U|| \cdot ||V|| \cdot \cos(\Theta) \), and because the projection of \( U \) onto \( V \) is given by \( ||U|| \cdot \cos(\Theta) \), we see that the projection of \( U \) onto \( V \) is actually \( U \cdot V \cdot ||V|| \). This is especially useful when \( V \) is a unit vector because then the dot product alone is the projection, and this is one of the reasons for normalizing the vectors we use.

![Figure 4.3: diagram of the projection of \( U \) onto \( V \)](image)

The second computation is the cross product, or vector product, of two vectors. The cross product of two vectors is a third vector that is perpendicular to each of the original vectors and whose length is the product of the two vector lengths times the sin of the angle between them. Thus if two vectors are parallel, the cross product is zero; if they are orthogonal, the cross product has length equal to the product of the two lengths; if they are both unit vectors, the cross product is the sine of the included angle. The computation of the cross product can be expressed as the determinant of a matrix whose first row is the three standard unit vectors, whose second row is the first vector of the product, and whose third row is the second vector of the product. Denoting the unit direction vectors in the X, Y, and Z directions as \( i \), \( j \), and \( k \), as above, we can express the cross product of two vectors \( <a,b,c> \) and \( <u,v,w> \) in terms of a determinant:

\[
< a,b,c > \times < u,v,w > = \begin{vmatrix} i & j & k \\ a & b & c \\ u & v & w \end{vmatrix} = \begin{vmatrix} i & j & k \\ b & c & 0 \\ u & v & w \end{vmatrix} - \begin{vmatrix} i & j & k \\ a & c & 0 \\ u & w & v \end{vmatrix} + \begin{vmatrix} i & j & k \\ a & b & 0 \\ u & v & w \end{vmatrix} = \left( bw - cv, cu - aw, av - bu \right)
\]

As an example here, consider the two points we saw earlier, but treat them as vectors: \( u = <3.0, 4.0, 5.0> \) and \( v = <5.0, -1.5, 4.0> \). Then the length of \( u \) is the square root of \( u \cdot u \), or 7.071, and the length of \( v \) is the square root of \( v \cdot v \), or 6.576. Then we see that \( u \cdot v = 15.0 - 6.0 + 20.0 = 29.0 \), and the cosine of the angle between \( u \) and \( v \) is \( 29.0/(7.071*6.576) \) or 0.624. Further, the cross product of the two vectors is computed as
\[
\begin{align*}
\mathbf{u} \times \mathbf{v} &= \det \begin{vmatrix}
\mathbf{i} & \mathbf{j} & \mathbf{k} \\
3 & 4 & 5 \\
5 & -1.5 & 4
\end{vmatrix} = \\
&= \mathbf{i} \begin{vmatrix} 4 & 5 \\ -1.5 & 4 \end{vmatrix} - \mathbf{j} \begin{vmatrix} 3 & 5 \\ 5 & -1.5 \end{vmatrix} + \mathbf{k} \begin{vmatrix} 3 & 4 \\ 5 & -1.5 \end{vmatrix}
\end{align*}
\]

Carrying out the 2x2 determinants gives us \( \mathbf{u} \times \mathbf{v} = 23.5 \mathbf{i} + 13.0 \mathbf{j} - 24.5 \mathbf{k} \) as the cross product. You should check to see that this product is orthogonal to both \( \mathbf{u} \) and \( \mathbf{v} \) by computing the dot products, which should be 0.

The cross product has a “handedness” property and is said to be a right-handed operation. That is, if you align the fingers of your right hand with the direction of the first vector and curl your fingers towards the second vector, your right thumb will point in the direction of the cross product. Thus the order of the vectors is important; if you reverse the order, you reverse the sign of the product (recall that interchanging two rows of a determinant will change its sign), so the cross product operation is not commutative. As a simple example, with \( \mathbf{i}, \mathbf{j}, \) and \( \mathbf{k} \) as above, we see that \( \mathbf{i} \times \mathbf{j} = \mathbf{k} \) but that \( \mathbf{j} \times \mathbf{i} = -\mathbf{k} \). In general, if you consider the arrangement of Figure 4.4, if you think of the three direction vectors as being wrapped around as if they were visible from the first octant of 3-space, the product of any two is the third direction vector if the letters are in counterclockwise order, and the negative of the third if the order is clockwise. Note also that the cross product of two collinear vectors (one of the vectors is a constant multiple of the other) will always be zero, so the geometric interpretation of the cross product does not apply in this case.

![Figure 4.4: the direction vectors in order](image)

The cross product can be very useful when you need to define a vector perpendicular to two given vectors; the most common application of this is defining a normal vector to a polygon by computing the cross product of two edge vectors. For a triangle as shown in Figure 4.5 with

![Figure 4.5: the normal to a triangle as the cross product of two edges](image)
vertices $A$, $B$, and $C$ in order counterclockwise from the “front” side of the triangle, the normal vector can be computed by creating the two difference vectors $P = C - B$ and $Q = A - C$, and computing the cross product as $P \times Q$ to yield a vector $N$ normal to the plane of the triangle. In fact, we can say more than this; the cross product of two vectors is not only perpendicular to the plane defined by those vectors, but its length is the product of their lengths times the sine of the angle between them. As we shall see in the next section, this normal vector, and any point on the triangle, allow us to generate the equation of the plane that contains the triangle. When we need to use this normal for lighting we will need to normalize it, or make it a unit-length vector as we described above, but that is easily done by calculating the vector’s length and dividing each component by that length.

**Reflection vectors**

There are several times in computer graphics where it is important to calculate a vector that is a reflection of another vector in some surface. One example is in specular light calculations; we will see later that the brightness of specular light at a point (shiny light reflected from the surface similarly to the way a mirror reflects light) will depend on the angle between the vector to the eye from that point and the reflection of the vector from that point to the light. Another example is in any model where objects hit a surface and are reflected from it, where the object’s velocity vector after the bounce is the reflection of its incoming velocity vector. In these cases, we need to know the normal to the surface at the point where the vector to be reflected hits the surface, and the calculation is fairly straightforward. Figure 4.6 shows the situation we are working with in the case of reflected velocities; the situation with light is similar except that the P vector is directed outwards instead of inwards.

![Figure 4.6: Incoming vector, outgoing vector, and normal vector](image)

In this figure, let $N^*$ be the vector that $Q$ makes when it is projected on $N$; then $N^* = -(N \cdot P)N$. We see that $X = P + N^* = P - (N \cdot P)N$. But $Q + P = 2X$, so $Q = 2(P - (N \cdot P)N) - P$, from which $Q = P - 2(N \cdot P)N$. This is actually an easy calculation and the code is left to the reader.

**Transformations**

In the previous two chapters we discussed transformations rather abstractly: as functions that operate on 3D space to produce given effects. In the spirit of this chapter, however, we describe how these functions are represented concretely for computation and, in particular, the representation of each of the basic scaling, rotation, and transformation matrices.

To begin, we recall that we earlier introduced the notion of homogeneous coordinates for points in 3D space: we identify the 3D point $(x, y, z)$ with the homogeneous 4D point $(x, y, z, 1)$. The transformations in 3D computer graphics are all linear functions on 4D space and so may be represented as 4x4 matrices:
Applying two transformations in order, or composing the transformations, is accomplished by multiplying the transformations’ matrices. So if we have transformations $S$ and $T$, represented by arrays $\{S[i][j]\}$ and $\{T[i][j]\}$, respectively, then the composition of the transformations is given by $C = S*T$; in terms of code for multiplying the matrices this is

```c
for (int i = 0; i < 4; i++)
  for (int j = 0; j < 4; j++)
    C[i][j] = 0.;
for (int k = 0; k < 4; k++)
  C[i][j] += S[i][k]*T[k][j];
```

This is straightforward but fairly slow; you may be able to find ways to speed it up. Geometrically, this treats the matrices as sets of vectors: the left-hand matrix is composed of row vectors and the right-hand matrix of column vectors. The product of the matrices is composed of the dot products of each row matrix from the left by each column matrix on the right.

The effect of a transformation on a vector is given by multiplying the transformation, as a matrix, on the left of the point, stored as a column vector. This uses the same code as above except that because we are multiplying a 4x4 matrix on the left by a 4x1 matrix on the right, the index $j$ has only the value 0 rather than the values 0 through 3.

So with this background, let’s proceed to consider how the basic transformations look as matrices. For scaling, the OpenGL function `glScalef(sx, sy, sz)` is expressed as

$$
\begin{bmatrix}
  sx & 0 & 0 & 0 \\
  0 & sy & 0 & 0 \\
  0 & 0 & sz & 0 \\
  0 & 0 & 0 & 1 \\
\end{bmatrix}
$$

For translation, the OpenGL function `glTranslatef(tx, ty, tz)` is expressed as

$$
\begin{bmatrix}
  1 & 0 & 0 & tx \\
  0 & 1 & 0 & ty \\
  0 & 0 & 1 & tz \\
  0 & 0 & 0 & 1 \\
\end{bmatrix}
$$

For rotation, we have a more complex situation. OpenGL allows you to define a rotation around any given line with the function `glRotatef(angle, x, y, z)` where `angle` is the amount or rotation (in degrees), and $\\langle x, y, z \rangle$ is the direction vector of the line around which the rotation is to be done. We can write a matrix for the general rotation, but before we do that, let’s look at the simpler rotations around the coordinate axes. For the rotation around the X-axis, `glRotatef(angle, 1., 0., 0.)`, the matrix is as follows; note that the first component, the X-component, is not changed by this matrix.

$$
\begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & \cos(angle) & -\sin(angle) & 0 \\
  0 & \sin(angle) & \cos(angle) & 0 \\
  0 & 0 & 0 & 1 \\
\end{bmatrix}
$$
For the rotation around the Z-axis, \texttt{glRotatef(angle, 0., 0., 1.)}, the matrix is much the same as the X-axis rotation but with a different fixed space.

\[
\begin{bmatrix}
\cos(\text{angle}) & -\sin(\text{angle}) & 0 & 0 \\
\sin(\text{angle}) & \cos(\text{angle}) & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

For the Y-axis, there is a difference because the cross produce of the X- and Z-axes is in the opposite direction to the Y-axis. This means that the angle relative to the Y-axis is the negative of the angle relative to the cross product, giving us a change in the sign of the sine function. So the matrix for \texttt{glRotatef(angle, 0., 1., 0.)} is:

\[
\begin{bmatrix}
\cos(\text{angle}) & 0 & \sin(\text{angle}) & 0 \\
0 & 1 & 0 & 0 \\
-\sin(\text{angle}) & 0 & \cos(\text{angle}) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

The formula for a rotation around an arbitrary line is more complex and is given in the OpenGL manual, so we will not present it here.

**Planes and half-spaces**

We saw above that a line could be defined in terms of a single parameter, so it is often called a one-dimensional space. A plane, on the other hand, is a two-dimensional space, determined by two parameters. If we have any two non-parallel lines that meet in a single point, we recall that they determine a plane that can be thought of as all points that are translations of the given point by vectors that are linear combinations of the direction vectors of the two lines. Thus any plane in space is seen as two-dimensional where each of the two lines contributes one of the dimensional components. In practice, we usually don’t have two lines in the plane but have three points in the plane that do not lie in a single straight line, and we get the two lines by letting each of two different pairs of points determine a line. Because each pair of points lies in the plane, so does each of the two lines they generate, and so we have two lines.

In more general terms, let’s consider the vector \( \vec{N} = \langle A, B, C \rangle \) defined as the cross product of the two vectors determined by the two lines. Then \( \vec{N} \) is perpendicular to each of the two vectors and hence to any line in the plane. In fact, this can be taken as defining the plane: the plane is defined by all lines through the fixed point perpendicular to \( \vec{N} \). If we take a fixed point in the plane, \((U, V, W)\), and a variable point in the plane, \((x, y, z)\), we can use the dot product to express the perpendicular relationship as

\[
\langle A, B, C \rangle \cdot \langle u-U, y-V, z-W \rangle = 0.
\]

When we expand this dot product we see

\[
A(u-X) + B(y-V) + C(z-W) = Ax + By + Cz + (-AU - BV - CW) = 0.
\]

This allows us to give an equation for the plane:

\[
Ax + By + Cz + D = 0
\]

for an appropriate value of \( D \). Thus the coefficients of the variables in the plane equation exactly match the components of the vector normal to the plane—a very useful fact from time to time.

We can readily see that a plane divides 3D space into two parts, but we need to know how to tell which points are in which part. To see this, let’s look first at 2D space. Any line divides a plane into two parts, and if we know the equation of the line in the traditional form

\[
ax + by + c = 0,
\]
then we can determine whether a point lies on, above, or below the line by evaluating the function \( f(x,y) = ax + by + c \) and determining whether the result is zero, positive, or negative, respectively. In a similar way, the equation for the plane as defined above does more than just identify the plane; it allows us to determine on which side of the plane any point lies. If we create a function of three variables from the plane equation

\[ f(x,y,z) = Ax + By + Cz + D, \]

then the plane consists of all points where \( f(x,y,z) = 0 \). All points \((x,y,z)\) with \( f(x,y,z) > 0 \) lie on one side of the plane, called the positive half-space for the plane, while all points with \( f(x,y,z) < 0 \) lie on the other, called the negative half-space for the plane. We will find that OpenGL uses the four coordinates \( A, B, C, D \) to identify a plane and uses the half-space concept to choose displayable points when the plane is used for clipping.

Let’s consider an example here that will illustrate both the previous section and this section. If we consider three points \( A = (1.0, 2.0, 3.0), B = (2.0, 1.0, -1.0), \) and \( C = (-1.0, 2.0, 1.0) \), we can easily see that they do not lie on a single straight line in 3D space. Thus these three points define a plane; let’s calculate the plane’s equation.

To begin, the difference vectors are \( A - B = <-1.0, 1.0, 4.0> \) and \( B - C = <3.0, -1.0, -2.0> \) for the original points, so these two vectors applied to any one of the points will determine two lines in the plane. We then compute the cross product \((B - C) \times (A - B)\) of these two vectors as we outlined above, and get

\[
\begin{vmatrix}
  i & j & k \\
  3 & -1 & -2 \\
  -1 & 1 & 4 \\
\end{vmatrix} = <-2.0, -10.0, 2.0> 
\]

Thus the equation of the plane is \(-2X - 10Y + 2Z + D = 0\), and putting in the coordinates of \( B \) we can calculate the constant \( D \) as \(-12.0\), giving a final equation as \(-2X - 10Y + 2Z - 12 = 0\). Here any point for which the plane equation yields a positive value lies on the side of the plane in the direction the normal is facing, and any point that yields a negative value lies on the other side of the plane.

**Distance from a point to a plane**

Just as we earlier defined a way to compute the distance from a point to a line, we also want to be able to compute the distance from a point to a plane. This will be useful when we discuss collision detection later, and may also have other applications.

![Figure 4.7: the computation of the distance from a point to a plane](image)

Let’s consider a plane \( Ax + By + Cz + D = 0 \) with normal vector \( N = \langle A, B, C \rangle \), unit normal vector \( n = \langle a, b, c \rangle \), an arbitrary point \( P = (u, v, w) \). Then select any point \( Q = (d, e, f) \) in the plane, and
consider the relationships shown in Figure 4.7. The diagram shows that the distance from the point to the plane is the projection of the vector $P - Q$ on the unit normal vector $n$. This gives us an easy way to compute this distance, especially since we can choose the point $Q$ any way we wish.

**Polygons and convexity**

Most graphics systems, including OpenGL, are based on modeling and rendering based on polygons and polyhedra. A *polygon* is a plane region bounded by a sequence of directed line segments with the property that the end of one segment is the same as the start of the next segment, and the end of the last line segment is the start of the first segment. A *polyhedron* is a region of 3–space that is bounded by a set of polygons. Because polyhedra are composed of polygons, we will focus on modeling with polygons, and this will be a large part of the basis for the modeling chapter below.

The reason for modeling based on polygons is that many of the fundamental algorithms of graphics have been designed for polygon operations. In particular, many of these algorithms operate by interpolating values across the polygon; you will see this below in depth buffering, shading, and other areas. In order to interpolate across a polygon, the polygon must be *convex*. Informally, a polygon is complex if it has no indentations; formally, a polygon is complex if for any two points in the polygon (either the interior or the boundary), the line segment between them lies entirely within the polygon.

Because a polygon bounds a region of the plane, we can talk about the interior or exterior of the polygon. In a convex polygon, this is straightforward because the figure is defined by its bounding planes or lines, and we can simply determine which side of each is “inside” the figure. If your graphics API only allows you to define convex polygons, this is all you need consider. In general, though, polygons can be non-convex and we would like to define the concept of “inside” for them. Because this is less simple, we look to convex figures for a starting point and notice that if a point is inside the figure, any ray from an interior point (line extending in only one direction from the point) must exit the figure in precisely one point, while if a point is outside the figure, if the ray hits the polygon it must both enter and exit, and so crosses the boundary of the figure in either 0 or 2 points. We extend this idea to general polygons by saying that a point is inside the polygon if a ray from the point crosses the boundary of the polygon an odd number of times, and is outside the polygon if a ray from the point crosses the boundary of the polygon an even number of times. This is illustrated in Figure 4.8. In this figure, points A, D, E, and G are outside the polygons and points B, D, and F are inside. Note carefully the case of point G; our definition of inside and outside might not be intuitive in some cases.

![Figure 4.8: Interior and exterior points of a convex polygon (left) and two general polygons (center and right)](image)
Another way to think about convexity is in terms of linear combinations of points. We can define a *convex sum* of points \( P_0, P_1, \ldots, P_n \) as a sum \( \sum c_i P_i \) where each of the coefficients \( c_i \) is non-negative and the sum of the coefficients is exactly 1. If we recall the parametric definition of a line segment, \( (1-t)P_0 + tP_1 \), we note that this is a convex sum. So if a polygon is convex, all convex sums of vertices of the polygon also lie in the polygon, which gives us an alternate definition of convex polygons that can sometimes be useful.

A convex polygon also has a broader property: any point in the polygon is a convex sum of vertices of the polygon. Because this means that the entire interior of the polygon can be expressed as a convex sum of the vertices, we would expect that interpolation processes such as depth (described in an earlier chapter) and color smoothing (described in a later chapter) could be expressed by the same convex sum of these properties for the vertices. Thus convexity is a very important property for geometric objects in computer graphics systems.

As we suggested above, most graphics systems, and certainly OpenGL, require that all polygons be convex in order to render them correctly. If you need to use a polygon that is not convex, you may always subdivide it into triangles or other convex polygons and work with them instead of the original polygon. As an alternative, OpenGL provides a facility to tessellate a polygon—divide it into convex polygons—automatically, but this is a complex operation that we do not cover in these notes.

**Polyhedra**

As we saw in the earlier chapters, polyhedra are volumes in 3D space that are bounded by polygons. In order to work with a polyhedron you need to define the polygons that form its boundaries. In terms of the scene graph, then, a polyhedron is a group node whose elements are polygons. Most graphics APIs do not provide a rich set of pre-defined polyhedra that you can use in modeling; in OpenGL, for example, you have only the Platonic solids and a few simple polyhedral approximations of other objects (sphere, torus, etc.) A convex polyhedron is one for which any two points in the object are connected by a line segment that is completely contained in the object.

Because polyhedra are almost always defined in terms of polygons, we will not focus on them but will rather focus on polygons. Thus in the next section when we talk about collision detection, the most detailed level of testing will be to identify polygons that intersect.

**Collision detection**

There are times when we need to know whether two objects meet in order to understand the logic of a particular scene, particularly when that scene involves moving objects. There are several ways to handle collision detection, involving a little extra modeling and several stages of logic, and we outline them here without too much detail because there isn’t any one right way to do it.

The first thing you must ask yourself is exactly what kind of collision you want to detect, and what objects in your model could collide. You will see shortly that there is a lot of logic involved in this process, and the two best ways to speed up the process are to avoid making tests when you can, and to make the simplest possible tests when you must test at all.

As we discuss testing below, we will need to know the actual coordinates of various points in 3D world space. You can track the coordinates of a point as you apply the modeling transformation to an object, but this can take a great deal of computation that we would otherwise give to the graphics API, so this works against the approach we have been taking. But your API may have the capability of giving you the world coordinates of a point with a simple inquiry function. In
OpenGL, for example, you can use the function glGetFloatv(GL_MODELVIEW_MATRIX) to get the current value of the modelview matrix at any point; this returns an array of 16 real values that is the matrix to be applied to your model at that point. If you treat this as a 4x4 matrix and multiply it by the coordinates of any vertex, you will get the coordinates of the transformed vertex in 3D eye space. This will give you a consistent space in which to make your tests as described below.

In order to simplify collision detection, it is usual to start thinking of possible collisions instead of actual collisions. Quick rejection of possible collisions will make a big difference in speeding up handling actual collisions. One standard approach is to use a substitute object instead of the real object, such as a sphere or a box that surrounds the object closely. These are called bounding objects, such as bounding spheres or bounding boxes, and they are chosen for the ease of collision testing. It is easy to see if two spheres could collide, because this happens precisely when the distance between their centers is less than the sum of the radii of the spheres. It is also easy to see if two rectangular boxes intersect because in this case, you can test the relative values of the larger and smaller dimensions of each box in each direction. Of course, you must be careful that the bounding objects are defined after all transformations are done for the original object, or you may distort the bounding object and make the tests more difficult.

As you test for collisions, then, you start by testing for collisions between the bounding objects of your original objects. When you find a possible collision, you must then move to more detailed tests based on the actual objects. We will assume that your objects are defined by a polygonal boundary, and in fact we will assume that the boundary is composed of triangles. So the next set of tests are for possible collisions between triangles. Unless you know which triangles in one object are closest to which triangles in another object, you may need to test all possible pairs of triangles, one in each object, so we might start with a quick rejection of triangles.

Just as we could tell when two bounding objects were too far apart to collide, we should be able to tell when a triangle in one object is too far from the bounding object of the other object to collide. If that bounding object is a sphere, you could see whether the coordinates of the triangle’s vertices (in world space) are farther from that sphere than the longest side of the triangle, for example, or if you have more detailed information on the triangle such as its circumcenter, you could test for the circumcenter to be farther from the sphere than the radius of the circumcircle. (The circumcenter of a triangle is the common intersection of the three perpendicular bisectors of the sides of the circle; see Figure 4.9 for a sketch that illustrates this.)

![Figure 4.9: the circumcenter and circumcircle of a triangle](image)

After we have ruled out impossible triangle collisions, we must consider the possible intersection of a triangle in one object with a triangle in the other object. In this case we work with each line segment bounding one triangle and with the plane containing the other triangle, and we compute the point where the line meets the plane of the triangle. If the line segment is given by the parametric equation \(Q_0 + t \cdot (Q_1 - Q_0)\) and let the plane of the triangle be \(Ax + By + Cz + D = 0\), we can
readily calculate the value of $t$ that gives the intersection of the line and the plane. If this value of $t$ is not between 0 and 1, then the segment does not intersect the plane and we are finished. If the segment does intersect the plane, we need to see if the intersection is within the triangle or not.

Once we know that the line is close enough to have a potential intersection, we move on to test whether the point where the line meets the plane inside the triangle, as shown in Figure 4.10. With the counterclockwise orientation of the triangle, any point on the inside of the triangle is to the left (that is, in a counterclockwise direction) of the oriented edge for each edge of the triangle. The location of this point can be characterized by the cross product of the edge vector and the vector from the vertex to the point; if this cross product has the same orientation as the normal vector to the triangle for each vertex, then the point is inside the triangle. If the intersection of the line segment and the triangle’s plane is $Q$, this means that we must have $N \cdot ((P1 - P0) \times (Q - P0)) > 0$ for the first edge, and similar relations for each subsequent edge.

![Figure 4.10: a point inside a triangle](image)

**Polar, cylindrical, and spherical coordinates**

Up to this point we have emphasized Cartesian, or rectangular, coordinates for describing 2D and 3D geometry, but there are times when other kinds of coordinate systems are most useful. The coordinate systems we discuss here are based on angles, not distances, in at least one of their terms. Because graphics APIs generally do not handle non-Cartesian coordinate systems directly, when you want to use them you will need to translate points between these forms and rectangular coordinates.

In 2D space, we can identify any point $(X, Y)$ with the line segment from the origin to that point. This identification allows us to write the point in terms of the angle $\Theta$ the line segment makes with the positive X-axis and the distance $R$ of the point from the origin as:

- $X = R \cos(\Theta)$, $Y = R \sin(\Theta)$ or, inversely,
- $R = \sqrt{X^2 + Y^2}$, $\Theta = \arccos(X / R)$

where $\Theta$ is the value between 0 and $2\pi$ that is in the right quadrant to match the signs of X and Y. This representation $(R, \Theta)$ is known as the *polar form* for the point, and the use of the polar form for all points is called the *polar coordinates* for 2D space. This is illustrated in the left-hand image in Figure 4.11.

There are two alternatives to Cartesian coordinates for 3D space. *Cylindrical coordinates* add a third linear dimension to 2D polar coordinates, giving the angle between the X-Z plane and the plane through the Z-axis and the point, along with the distance from the Z-axis and the Z-value of the point. Points in cylindrical coordinates are represented as $(R, \Theta, Z)$ with $R$ and $\Theta$ as above and
with the Z-value as in rectangular coordinates. The right-hand image of Figure 4.11 shows the structure of cylindrical coordinates for 3D space.

![Figure 4.11: polar coordinates (left) and cylindrical coordinates (right)](image)

Cylindrical coordinates are a useful extension of a 2D polar coordinate model to 3D space. They not particularly common in graphics modeling, but can be very helpful when appropriate. For example, if you have a planar object that has to remain upright with respect to a vertical direction, but the object has to rotate to face the viewer in a scene as the viewer moves around, then it would be appropriate to model the object’s rotation using cylindrical coordinates. An example of such an object is a billboard, as discussed later in the chapter on high-efficiency graphics techniques.

**Spherical coordinates** represent 3D points in terms much like the latitude and longitude on the surface of the earth. The latitude of a point is the angle from the equator to the point, and ranges from 90° south to 90° north. The longitude of a point is the angle from the “prime meridian” to the point, where the prime meridian is determined by the half-plane that runs from the center of the earth through the Greenwich Observatory just east of London, England. The latitude and longitude valued uniquely determine any point on the surface of the earth, and any point in space can be represented relative to the earth by determining what point on the earth’s surface meets a line from the center of the earth to the point, and then identifying the point by the latitude and longitude of the point on the earth’s surface and the distance to the point from the center of the earth. Spherical coordinates are based on the same principle: given a point and a unit sphere centered at that point, with the sphere having a polar axis, determine the coordinates of a point $P$ in space by the latitude $\Phi$ (angle north or south from the equatorial plane) and longitude $\Theta$ (angle from a particular half-plane through the diameter of the sphere perpendicular to the equatorial plane) of the point where the half-line from the center of the sphere, and determine the distance from the center to that point. Then the spherical coordinates of $P$ are $(R, \Theta, \Phi)$.

Spherical coordinates can be very useful when you want to control motion to achieve smooth changes in angles or distances around a point. They can also be useful if you have an object in space that must constantly show the same face to the viewer as the viewer moves around; again, this is another kind of billboard application and will be described later in these notes.

It is straightforward to convert spherical coordinates to 3D Cartesian coordinates. Noting the relationship between spherical and rectangular coordinates shown in Figure 4.9 below, and noting that this figure shows the Z-coordinate as the vertical axis, we see the following conversion equations from polar to rectangular coordinates.

\[
x = R \cos(\Phi) \sin(\Theta) \\
y = R \cos(\Phi) \cos(\Theta) \\
z = R \sin(\Phi)
\]
Converting from rectangular to spherical coordinates is not much more difficult. Again referring to Figure 4.9, we see that \( R \) is the diagonal of a rectangle and that the angles can be described in terms of the trigonometric functions based on the sides. So we have the equations

\[
R = \sqrt{X^2 + Y^2 + Z^2} \\
\Phi = \arcsin \left( \frac{Z}{R} \right) \\
\Theta = \arctan \left( \frac{X}{\sqrt{X^2 + Y^2}} \right)
\]

Note that the inverse trigonometric function is the principle value for the longitude (\( \Phi \)), and the angle for the latitude (\( \Theta \)) is chosen between 0° and 360° so that the sine and cosine of \( \Theta \) match the algebraic sign (+ or −) of the X and Y coordinates.

Figure 4.12 shows a sphere showing latitude and longitude lines and containing an inscribed rectangular coordinate system, as well as the figure needed to make the conversion between spherical and rectangular coordinates.

Figure 4.12: spherical coordinates (left); the conversion from spherical to rectangular coordinates (right)

Higher dimensions?

While our perceptions and experience are limited to three dimensions, there is no such limit to the kind of information we may want to display with our graphics system. Of course, we cannot deal with these higher dimensions directly, so we will have other techniques to display higher-dimensional information. There are some techniques for developing three-dimensional information by projecting or combining higher-dimensional data, and some techniques for adding extra non-spatial information to 3D information in order to represent higher dimensions. We will discuss some ideas for higher-dimensional representations in later chapters in terms of visual communications and science applications.