Vector Geometry for Computer Graphics

James R. Miller
University of Kansas

Implementing and using computer graphics and modeling systems rely on mathematical operations on points and vectors. This tutorial reviews the basic ideas surrounding points and vectors in affine and projective spaces. In Part 1 of this tutorial I advocate vector geometric analysis to simplify required derivations and implementations. Part 2 in the next issue builds on this background and shows several specific applications of these techniques.

If you had a lump of clay and wanted to make a bowl, you would most likely fashion the clay by translating your abstract notions of desired size and shape directly into the appropriate tugs and pulls. You would probably not think much about coordinate systems.

If you wanted to make a table on which to display your bowl, and then some chairs so people could sit around the table, you would again think at a fairly high level of abstraction about appropriate overall sizes and desired spatial relationships. Even though you would alternately think about how a leg fits on a chair and where each completed chair should stand around the table, these thoughts would not involve "chair coordinate systems" and "table coordinate systems," and how one relates to the other.

Computer graphics and modeling rely on mathematical operations on points and vectors. I advocate using vector geometric analysis to simplify required derivations, as explained here.

On the other hand, when we want to use computers to help us design and visualize the objects in our world, we need a formal way to communicate our notions of size and shape to the computer. Obviously, coordinate systems come into play here, and we all know how they work.

In fact, coordinate systems work so well in this context that most graphics systems use all sorts of them. We would model component objects like legs in *local* coordinate systems. These coordinate systems are then instantiated multiple times inside of a chair local coordinate system; each chair system goes into the table's local coordinate system. This process of instantiating an object defined in one local coordinate system inside of another local coordinate system continues until we have defined the entire scene. This "special" final local coordinate system is commonly called the *world* coordinate system.

After having constructed the geometry in this fashion, we establish another coordinate system (often called an *eye* coordinate system) to describe how we want to look at our table and chairs. And we typically use yet other coordinate systems as well: projection coordinate systems, device coordinate systems, and so forth.

Clearly we must become proficient in using coordinate systems and comfortable with issues of how they relate to one another.

We will study a common set of such coordinate systems as well as transformations between them, but first a caveat appropriate to the primary theme of this article: we must remember that coordinate systems are merely an artifact of the language we must use to communicate with a computer. That is, our base ideas of size, shape, and orientation are intrinsic properties of the objects we create and remain independent of any coordinate system. We want to use coordinate systems where appropriate and necessary, but we don't want our thinking or analytical processes to become unnecessarily dependent upon coordinate systems. This probably doesn't make much sense just yet. As you will see while studying the mathematics underlying computer graphics operations, translating our original abstract notions of object manipulation, viewing specifications, and so forth into concrete computer-based actions can become overly complex if we force ourselves to think solely in terms of describing these actions relative to some specific coordinate system.

I describe as *vector geometric* that class of techniques based on representing and manipulating intrinsic relationships between objects that are independent of any coordinate system, such as the centroid of a group of points or the vector normal to two others. By contrast, *coordinate-based* approaches generally operate by comparing and manipulating individual x, y, and z coordinates of points. For example, a particular algorithm may select one of two points based on whose z coordinate is larger. You will come to understand and appreciate these distinctions more completely later. It will become apparent that vector geometric techniques prove most suitable when operating on objects whose position and orientation with respect to the current coordinate system are completely general, and when

0272-1716/99/$10.00 © 1999 IEEE

no axis of the coordinate system has any special relationship to the current problem of interest. Coordinate-based methods become preferable when we either know a priori that the geometry occupies a known, simple position and orientation in the coordinate system, or we preprocess it so that it does.

In this article I present the basic language and methods of vector geometric analysis and characterize the situations favoring vector geometric versus coordinate-based analysis. A companion article (Part 2 of this tutorial)[1] illustrates many of the ideas by deriving specific expressions and computer code for common important operations. Taken together, these two articles represent a condensed version of an earlier technical report with expanded applications.[2]

A set of C++ classes I developed includes methods and definitions of overloaded operators implementing nearly all of the low-level operations and matrix construction algorithms presented in this article and its companion.[1] Sample code using these tools to perform representative common operations appears in both the companion article and technical report.[1,2]

Points and vectors in affine and projective spaces

This section reviews many basic yet critical concepts related to points and vectors, and the spaces in which they live. I stress developing a geometric intuition that will help you understand the derivations that follow and master the tools required to develop more advanced vector geometric analysis abilities.

I limit the treatment of these topics—especially those related to projective spaces and projective maps—to that required to support a graphics pipeline. Several good references cover other perspectives as well as additional detail on this material (see the "Further Reading" sidebar).

In this section we begin by developing a formal understanding of points and vectors. In a sense, they are "performers in a play." Next I'll briefly discuss vector, affine, and projective spaces. Extending the analogy, spaces represent the "sets" in which our "performers" operate. Finally, we consider various transformations, maps, and other operations applicable to points and vectors in the various spaces. These operations characterize how our "performers" work with one another and move about within a given "set."

A thorough understanding of this material is important because, in all likelihood, no graphics or modeling system will provide all the functionality you require in exactly the way you need it. Inevitably, the need will arise for you to derive and implement computations to obtain some required geometric quantity. If you cannot justify your derivations in terms of the properties and allowed operations described in this section, your derivation will most likely prove invalid.

Points and vectors

We all have an intuitive sense of points and vectors. A point is simply a position in space. To describe and manipulate a point numerically, we must "address" it by specifying signed distances from some reference point along linearly independent directions (that is, we must assign it "coordinates"), but the point stays the same regardless of the coordinate system we use to address it.

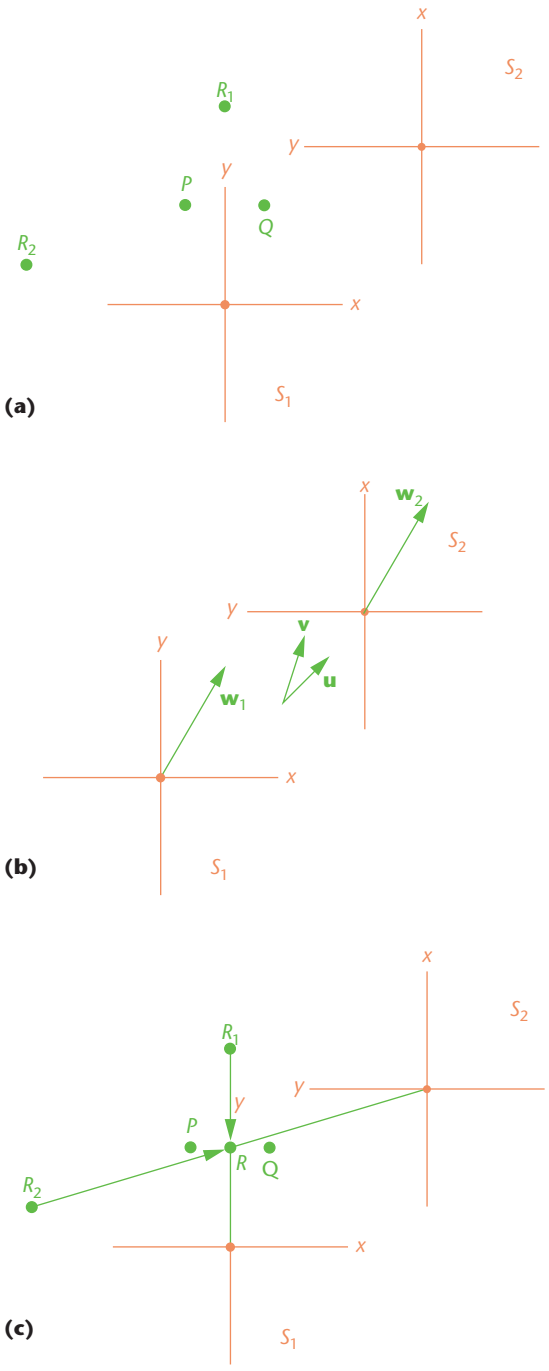
As a concrete example, the swing set in my back yard sits at a fixed location. From the door to my back yard, my kids go 20 feet west and 25 feet north to get to it; hence its coordinates with respect to the coordinate system of my back door are (–20, 25). My neighbor's kids also like to play on our swing set, but from their back door, they go 30 feet south and 120 feet west. Hence the swing set's coordinates with respect to the coordinate system of their back door are (–120, –30). Certainly the

Further Reading

Goldman[1] provides a rigorous theoretical justification for many of the concepts that I introduce at a more intuitive level here. He also gives several additional examples of vector geometric expressions relating to useful curve and surface properties and analytical queries. Goldman[2] and DeRose[3] have advanced very similar recommendations for geometric software development based on the rigorous use of the properties of points and vectors in affine spaces as described here. DeRose[3] in addition suggests that homogeneous coordinates (projective space points) are overused in graphics systems. He argues that the simpler affine formulations are often preferable. Several standard textbooks address these issues as well. While it is impractical to provide an exhaustive list of related textbooks, examples of standard graphics texts with relevant material include the appendix of *Computer Graphics: Principles and Practice*,[4] appendices B and C of *Interactive Computer Graphics: A Top-Down Approach With OpenGL*,[5] appendix A of *Computer Graphics*,[6] and chapter 7 of *Computer Graphics*.[7] Such material tends to be covered earlier and used more extensively in many texts on curves and surfaces. Again, an exhaustive list is impractical, but examples include chapters 1 and 2 of *NURB Curves and Surfaces: From Projective Geometry to Practical Use*[8] and chapter 2 of *Curves and Surfaces for Computer-Aided Geometric Design*.[9]

References

1. R.N. Goldman, "Illicit Expressions in Vector Algebra," *ACM Trans. on Graphics*, Vol. 4, No. 3, July 1985, pp. 223-243.
2. R.N. Goldman, "Vector Geometry: A Coordinate-Free Approach," *Siggraph 85 Short Course No. 16*, ACM Press, New York, July 1985.
3. T. DeRose, "A Coordinate-Free Approach to Geometric Programming," *Contemporary Approaches to Geometry for Computer Graphics and Computer-Aided Design*, Siggraph 89 Short Course No. 14, ACM Press, New York, Aug. 1989. (A condensed version also appeared in *Theory and Practice of Geometric Modeling*, W. Strasser and H. Seidel, eds., Springer Verlag, Berlin, 1989, pp. 291-306.)
4. J.D. Foley et al., *Computer Graphics: Principles and Practice*, Addison-Wesley, Reading, Mass., 1990.
5. E. Angel, *Interactive Computer Graphics: A Top-Down Approach With OpenGL*, Addison-Wesley, Reading, Mass., 1997.
6. D. Hearn and M.P. Baker, *Computer Graphics*, Prentice Hall, Englewood, N.J., 2d ed., 1994.
7. F.S. Hill, Jr., *Computer Graphics*, Macmillan Publishing, New York, 1990.
8. G. Farin, *NURB Curves and Surfaces: From Projective Geometry to Practical Use*, A.K. Peters, Natick, Mass., 1995.
9. G. Farin, *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*, 4th edition, Academic Press, San Diego, 1996.



1 Candidate arithmetic operations on points and vectors. (a) Addition of points is not well defined. (b) Addition of vectors is well defined. (c) Midpoint is an exception.

swing set exists independent of either of our back doors; the doors merely provide two equally convenient reference frames to describe the location of the swing set. Similarly, points do not "belong" in any sense to any coordinate system; we merely describe their positions by reference to such systems.

By contrast, a vector has no location. Rather, it's defined solely as a direction and a length. Like the point, we describe its direction numerically with respect to some coordinate system, but—also like the point—this numeric description differs in different coordinate systems.

The fact that a vector has no position means that we can draw the visual representation of a vector anywhere. We can also drag such a visual representation anywhere we wish.

One advantage of carefully preserving the distinction between points and vectors is that we can derive very general expressions and algorithms for important operations independent of the geometry's position and orientation with respect to any particular coordinate system. While generality often comes with increased overhead and cost, it turns out in this case that computer code based on vector geometric analyses often exhibits improved computational speed, better numerical reliability, and simplicity of implementation. I'll give examples supporting these claims as we proceed through this article and its companion.[1]

Considering the example below gives a quick sense for why we must distinguish between points and vectors. We'll see that addition of points is meaningless, whereas addition of vectors is meaningful (not to mention useful).

In Figure 1a, notice two points (P and Q) and two reference coordinate systems (S_1 and S_2). As measured in S_1, $P = (-1, 2.5)$ and $Q = (1, 2.5)$. Similarly, in S_2, $P = (-1.5, 6)$ and $Q = (-1.5, 4)$. So what do we get if we add P and Q? Using their representation in S_1, we get $R_1 = P + Q = (0, 5)$. Using their representation in S_2, we get $R_2 = P + Q = (-3, 10)$. What's going on? The results differ. (We needed two different dots for R_1 and R_2.) So what does this mean? Well, we started with two points in space, a concept independent of any coordinate system. Then we added the points and obtained answers that depended on which coordinate system we used to describe them. This is exactly the problem.

To be well defined, an operation must produce results independent of any coordinate system when given inputs that are coordinate system independent. Clearly, addition of points fails this test. It's easy to construct similar examples illustrating why multiplication of points by arbitrary scalars (such as "2 * P") is meaningless.

Now let's see how these problems do not exist with vectors. Figure 1b shows the same two coordinate systems with two vectors $\mathbf{u}$ and $\mathbf{v}$. As measured in S_1, the vector $\mathbf{u} = (1, 1)$ and $\mathbf{v} = (0.5, 1.5)$. Similarly, in S_2, $\mathbf{u} = (1, -1)$ and $\mathbf{v} = (1.5, -0.5)$. So what do we get if we add $\mathbf{u}$ and $\mathbf{v}$? Using their representation in S_1, we get $\mathbf{w}_1 = \mathbf{u} + \mathbf{v} = (1.5, 2.5)$. Using their representation in S_2, we get $\mathbf{w}_2 = \mathbf{u} + \mathbf{v} = (2.5, -1.5)$. As with the previous example, the numbers differ. However, unlike the previous example, the vector is the same in either case. To see this, drag the tail of the visual representation of $\mathbf{w}_1$ to the tail of that of $\mathbf{w}_2$. (Recall that since vectors have no position—they describe a relative quantity—we can drag their visual representations wherever we wish.) The vectors are the same; that is, we get the same result when using either (or, in general, any) coordinate system.

In summary, what we get when we add two points or multiply a point by a scalar depends on the coordinate system in which we choose to work. This is bad. What we get when we add two vectors or multiply a vector by a scalar remains independent of the coordinate system in which we choose to work. This is good. Together these two observations demonstrate that points and vectors are indeed different concepts and should be treated as

such. Many other examples will show the importance of differentiating between points and vectors.

We cannot quite leave this example yet, because it leaves us with a problem. Consider what happens if we divide R_1 and R_2 in the example above by two. We get $R_1/2 = (0, 2.5)$ in S_1, and $R_2/2 = (-1.5, 5)$ in S_2. These two points—labeled R in Figure 1c—are now the same. (As with the vector example, the coordinates still differ, but the points are indeed the same.) Of course, this should not surprise us. Adding two points and then dividing by two gives us the midpoint, an operation certainly well defined independent of any coordinate system.

Thus we seem to have two conflicting results: since addition and scalar multiplication are not well defined in general, it would appear that our midpoint operation cannot be justified. But of course we know that it can be. So the question arises, what is special about the midpoint calculation? And do other "special" operations exist? How can we characterize them in general?

To address these and other questions, we must formalize our understanding of the spaces in which points and vectors live. In the following section, I therefore introduce vector spaces, affine spaces, and others. Once you have an understanding of these spaces and the operations defined in them, you can answer the questions just raised.

Spaces

So far we have focused on points and vectors—our performers—but not the spaces in which they live. Turning our attention to these spaces, we will immediately see other differences between points and vectors. We will also see new ways to understand the relationships between them.

> *Definition*: An n-dimensional *vector space* consists of a set of vectors and two operations: addition and scalar multiplication. The vector space is closed under these two operations: addition of two vectors yields a vector in the vector space; multiplication of a vector by a scalar also produces a vector in the vector space. Finally, there exists a distinguished member of the set called the *zero vector* $\mathbf{0}$ with the properties that $a \cdot \mathbf{0} = \mathbf{0}$ for all scalars a, and $\mathbf{0} + \mathbf{v} = \mathbf{v} + \mathbf{0} = \mathbf{v}$ for all vectors $\mathbf{v}$.

We can interpret adding two vectors visually by positioning the tail of one at the head of the other. The vector from the tail of the first to the head of the second represents the vector sum. Multiplying a vector by a scalar scales the length of the vector by the absolute value of the scalar. The direction does not change for positive scale factors; the direction flips for negative scale factors.

> *Definition*: An n-dimensional *affine space* consists of a set of points, an associated n-dimensional vector space, and two operations: subtraction of two points in the set and addition of a point in the set and a vector in the associated vector space. The former produces a vector in the associated vector space, and the latter produces another point in the affine space. Unlike a vector space, which has the distinguished vector $\mathbf{0}$, an affine space does not include a distinguished point.

Notice that this definition does not permit addition of points or multiplication of points by arbitrary scalars. This should not surprise us—the examples from the previous section illustrated that such operations were not well defined, since they produced coordinate-system-dependent results.

Armed with these two definitions, let's now return to the questions left dangling at the end of the previous section. At first glance, the results of the examples we considered there would lead us to believe that for arbitrary scalars, points, and vectors (a_i, P_i, and $\mathbf{v}_i$, respectively),

$$\sum_{i=0}^{n} a_i P_i = \text{undefined}$$

but

$$\sum a_i \mathbf{v}_i = \text{vector}$$

The problem is that we know of one specific example where such a linear combination of points is valid: midpoint computation. To resolve this apparent problem, let us rewrite our problematic linear combination of points:

$$\begin{aligned}\sum_{i=0}^{n} a_i P_i &= \sum_{i=0}^{n} a_i P_0 + \sum_{i=0}^{n} a_i (P_i - P_0) \\ &= \alpha P_0 + \sum_{i=0}^{n} a_i (P_i - P_0) \end{aligned} \quad (1)$$

where

$$\alpha = \sum_{i=0}^{n} a_i$$

is a constant.

From the definition of affine spaces, we know that subtracting two points yields a well-defined vector. Therefore the summation on the second line of Equation 1 will always yield a well-defined vector, since it's just a linear combination of vectors. Hence the original summation is well defined if and only if we choose α such that αP_0 is well defined. (Recall I said earlier only that multiplication of points by arbitrary scalars was not well defined. Perhaps we can find particular scalars that produce well-defined results when used to multiply points. Indeed we can.)

Observe that if $\alpha = 1$, αP_0 is well defined in the sense that we have considered so far. That is, $1 \cdot P_0$ yields the same point regardless of the coordinate system in which we measure P_0. Since midpoint computation is simply an operation in which $n = 1$ and $a_0 = a_1 = 1/2$ (that is, $\alpha = 1$), we now understand why that operation "works."

Next observe that if $\alpha = 0$, αP_0 yields the n-tuple $(0, \ldots, 0)$. If we interpret this n-tuple as a point at the origin, then the result obviously depends on the coordinate system, hence it's meaningless. However, if we instead interpret this n-tuple as the zero vector in the associated vector space, the result remains independent of the coor-

dinate system. Therefore, we say that multiplying a point by "0" yields the zero vector, independent of the coordinate system used. In the context of Equation 1, then, the second line takes the form "vector + vector" when $\alpha = 0$.

No other choices exist for α. That is, for a scalar α and a point P, we can prove that αP is well defined if and only if α equals either 0 or 1. I leave the proof as an exercise.

Collecting these facts, we can now state that

$$\sum_{i=0}^{n} a_i P_i = \begin{cases} \text{point} & \text{if } \sum_{i=0}^{n} a_i = 1 \\ \text{vector} & \text{if } \sum_{i=0}^{n} a_i = 0 \\ \text{undefined} & \text{otherwise} \end{cases} \quad (2)$$

Not only does this characterization tell us what's allowed in terms of expressions involving points and vectors, it also gives us a check of sorts for derivations we perform. That is, we can check the "reasonableness" of some derivation by summing the coefficients on all the points in the expression. If they do not sum identically to 0 or 1, our formula cannot be correct. If they do, then we know the result is, respectively, a vector or a point. Based on this characterization, for example, we can state that, given points P, Q, R, and S, "$2P - Q - R + S$" describes a well-defined point, but "$P - Q + R + S$" is undefined.

Note that we have switched fairly casually between two subtly different types of arithmetic operations. Our definitions specify an allowed type: vector spaces permit addition and scalar multiplication ($\mathbf{u} + \mathbf{v}$ and $a\mathbf{v}$); affine spaces define subtraction of points and addition of a point and a vector ($P - Q$ and $P + \mathbf{v}$). When we write one of the two valid forms of the second line of Equation 1,

$$P_0 + \sum_{i=0}^{n} a_i (P_i - P_0) \quad (3)$$

or

$$\sum_{i=0}^{n} a_i (P_i - P_0) \quad (4)$$

we use only these permitted operations. That is, every individual instance of an addition, subtraction, or multiplication operator in Equations 3 and 4 is formally justifiable from the definitions. However, when we write the algebraically equivalent summation on the left-hand side of the first line of Equation 1 (or any specific example thereof, like "$2P - Q - R + S$"), we employ individual operations that, technically speaking, aren't permitted. For example, the multiplication of P by 2 isn't by itself a valid operation. Yet the expression as a whole is well defined.

So what are the practical implications of this? As graphics and modeling programmers, we generally derive geometric expressions that we then implement in our programs. It's typically inconvenient to express desired formulas using only the "sanctioned" affine and vector space operations, and we would certainly not want our programming tools to force us to do so. However, this implies that the onus stays on us to ensure that expressions we implement in computer code remain valid from the perspective of Equation 2.

DeRose formalized this concept in a geometric programming language that systematically performs various kinds of type checking using this characterization.[3] A broader and deeper theoretical treatment of this material appears elsewhere.[4,5]

Now let's resume our study of spaces. I stated earlier that a direction and a length characterized a vector, yet no operation or other mechanism in a vector space provides us with an absolute measure of length. (We can only say that one vector's length equals some multiple of another's length.) Furthermore, no tools associated with affine spaces allow us to talk about the strongly related notion of the absolute distance between two points. Euclidean spaces remedy this situation.

Definition: A *Euclidean space* is an affine space with the additional concept of distance.

The primary additional operation applicable to vectors in a Euclidean space is the dot product (or inner product). I'll discuss the definition, properties, and various applications of the dot product later. Here I simply point out that we can define the length of a vector $\mathbf{u}$ in terms of the dot product as

$$|\mathbf{u}| = \sqrt{\mathbf{u} \cdot \mathbf{u}}$$

(The vertical bars around $\mathbf{u}$ denote "length of $\mathbf{u}$".) Since we can characterize the distance between two points as the length of the vector between them, the dot product also gives us a way to measure distances between points in an affine space. Specifically, the distance between points P and Q is

$$|P - Q| = \sqrt{(P - Q) \cdot (P - Q)}$$

We all grew up with affine and Euclidean spaces and feel reasonably comfortable with them. Now let's get uncomfortable. Imagine that the affine space we know is actually a plane inside of a space with a dimension of one greater than the affine space. For example, consider the conventional 2D xy affine plane; suppose that it's really the $w = 1$ plane of a 3D xyw coordinate space. We call this "larger" space a *projective space* and say that our affine space is embedded in the projective space. (This relationship superficially resembles the way that 2D affine space relates to 3D affine space. That is, the xy coordinate plane simply lives on the $z = 0$ plane of 3D space. However, the superficial similarity ends there, as you'll see.)

The following definitions formalize these notions.

Definition: An $(n + 1)$*-dimensional projective space* is the space in which the points of an n-dimensional affine space are embedded. We denote the extra coordinate dimension as w and say that the affine points lie in the $w = 1$ plane of the projective space.

Definition: All projective space points on the line from the projective space origin through an affine point on the $w = 1$ plane are said to be *projectively equivalent* to the affine space point.

Returning to 2D affine space as an example, every point (x, y) in the affine space is actually a point $(x, y, 1)$ in the projective space. (See Figure 2.) Furthermore, all projective space points (wx, wy, w) where $w \neq 0$ are projectively equivalent to the point (x, y). Stated in another way, corresponding to every point (x, y) in affine space there exists the line in projective space that passes through $(0, 0, 0)$ and $(x, y, 1)$.

The same relationship holds between 3D affine space and projective *xyzw* space. That is, our 3D world lives on the $w = 1$ plane of projective *xyzw* space. Obviously, this is harder to visualize. Classical references for trying to understand geometry in dimensions greater than three include the nineteenth century book *Flatland*[6] and more recent publications.[7,8]

To this point, we have considered general n-dimensional spaces, using 2D and 3D spaces as specific examples. For the remainder of this tutorial, we'll focus on 3D vector and affine spaces—the domain of traditional computer graphics systems.

Affine maps

We have seen the importance of coordinate system transformations to the internal operation of graphics systems. We can view such a transformation as a function X that maps a point's coordinates as measured in one coordinate system into that point's coordinates as measured in another.

In dealing with points in affine spaces, we restrict X to the family of affine maps. X is an affine map if it takes the form

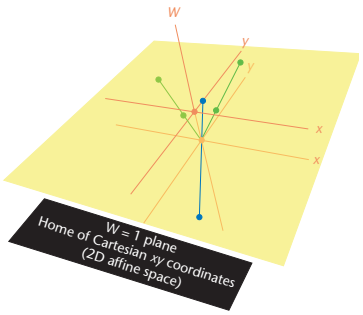
$$X(Q) = \mathbf{M}Q + \mathbf{t}$$

where $\mathbf{M}$ represents a 3×3 matrix and $\mathbf{t}$ a vector in the associated vector space.[9] We can use any combination of translation, rotation, scaling, and shear in an affine transformation, and it's well known that you can compose any affine map using some combination of these.[9]

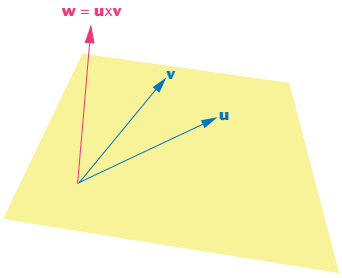
Affine maps preserve parallelism (for example, two parallel line segments remain parallel following transformation by an affine map), and they preserve ratios of signed distances between collinear points.[9] As a consequence of these properties, it's meaningful to apply the 3×3 matrix $\mathbf{M}$ of an affine map to a vector $\mathbf{w}$ in the associated vector space. To see this, choose an arbitrary pair of points P and Q such that $\mathbf{w} = P - Q$. Now if we apply our affine transformation individually to P and Q, then form the vector between the resulting transformed points, we obtain

$$\begin{aligned} X(P) - X(Q) &= (\mathbf{M}P + \mathbf{t}) - (\mathbf{M}Q + \mathbf{t}) \\ &= \mathbf{M}(P - Q) = \mathbf{M}\mathbf{w} \end{aligned}$$

Hence, to apply an affine transformation to a vector in the associated vector space, we ignore $\mathbf{t}$ and multiply the vector by $\mathbf{M}$.



2 An affine space embedded in a projective space.



3 The cross product, w, of vectors u and v.

Common vector operations

Two particularly powerful operations that come with Euclidean spaces are the dot product and the cross product of vectors. We'll use these operations extensively in our derivations.

For two arbitrary vectors $\mathbf{u}$ and $\mathbf{v}$, we define their dot product as the scalar

$$\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}|\,|\mathbf{v}| \cos\theta$$

where θ $(0 \leq \theta \leq \pi)$ represents the angle between the two vectors when drawn "tail to tail." Given two specific vectors $\mathbf{u}$ and $\mathbf{v}$, we compute a numerical value for the dot product as

$$\mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y + u_z v_z$$

For two arbitrary vectors $\mathbf{u}$ and $\mathbf{v}$, we define their cross product as a vector whose length is

$$|\mathbf{u} \times \mathbf{v}| = |\mathbf{u}|\,|\mathbf{v}| \sin\theta$$

We define the *direction* of the cross product vector as perpendicular to both $\mathbf{u}$ and $\mathbf{v}$ with sense determined by the right-hand rule. (See Figure 3.) Given two specific vectors $\mathbf{u}$ and $\mathbf{v}$, we compute a numerical value for the cross product using determinants as

$$\mathbf{u} \times \mathbf{v} = \left(\begin{vmatrix} u_y & u_z \\ v_y & v_z \end{vmatrix}, -\begin{vmatrix} u_x & u_z \\ v_x & v_z \end{vmatrix}, \begin{vmatrix} u_x & u_y \\ v_x & v_y \end{vmatrix} \right)$$

Coordinate versus vector geometric approaches to analysis

Before considering how we apply the tools we've seen to this point, I'll briefly review two major approaches to the mathematical analysis employed in graphics and modeling systems: *coordinate-based* and *vector geometric*. While both approaches have sufficient power for our needs, you'll see that they have complementary strengths. Each will emerge as ideal for different situations.

Coordinate-based methods refer to mathematical analyses based directly on the relationship of geometry to some specific coordinate system. For example, I might ask how the z coordinates of two points compare. By contrast, vector geometric methods focus on relation-

ships between geometric entities, not their individual positions and orientations with respect to some particular coordinate system. For example, I might compute the vector between two points and find its length in the direction of some other relevant unit vector.

You may argue at this point: "But how can you compute the vector and find its length in the specified direction without using coordinates of the points?" It's absolutely correct that to carry out these operations on a computer, you need to use coordinates. However, the level of abstraction used in the mathematical analysis remains focused on points and vectors in Euclidean spaces. We'll see several examples of the benefit of this higher level of abstraction as we proceed. For example, coordinate-based approaches to constructing matrices for general rotations must employ numerically sensitive tests to determine whether intermediate floating-point values are zero or close to it. By contrast, the vector-geometric approach that we'll derive requires only one initial test to ensure that the programmer has not given us a zero vector as a rotation axis. Not only will we need just this one test, but it's also easier to make it numerically robust.

With respect to a typical graphics pipeline, we find that vector geometric approaches work best when deriving transformations for the pipeline's early stages (such as various modeling and view orientation transformations). But then the situation changes. The view orientation transformation is typically constructed so as to produce a viewing coordinate system that possesses certain important relationships with respect to the graphics display device. Not only do these relationships simplify (both for the end user and the graphics programmer) the subsequent specification of projection data, they also can be exploited to simplify the analysis required to develop subsequent transformations and other viewing operations. Hence, we can accomplish this subsequent analysis most expediently using coordinate-based schemes that exploit the special properties "built into" these viewing coordinate systems.

Coordinate-based approaches

Coordinate-based approaches in effect force us to use a particular coordinate system as an intermediary when asking questions about the relationship between two objects. For example, instead of directly specifying a query between two entities A and B, I must first look at property p of A with respect to the coordinate system and compare the result to property p of B with respect to the same coordinate system. Based on the comparison, I must try to determine an answer to the original question. If I'm lucky—if A and B happen to be described in a convenient coordinate system—everything's OK. Otherwise it gets tricky.

When this approach works, it generally works well. As an example, suppose we have transformed two points into an eye coordinate system, and now we need to know which is closer to the eye so that we'll know which is visible. The eye coordinate system is constructed so that distance from the eye relates directly to the z coordinate. Hence we use the z coordinate as property p and can answer our original question simply by comparing the two z coordinates. Nothing could be simpler.

On the other hand, suppose we have two arbitrary vectors **u** and **v**, and want the matrix that rotates **u** onto **v**. Since we can't assume any special relationships among **u**, **v**, and the coordinate system, a coordinate-based approach would have to determine the series of transformations that would map each vector onto, say, the z axis. The desired transformation would then result from concatenating the transformations for **u** followed by the inverse of the transformations for **v**. This gets even more complicated than it sounds to implement correctly.[2]

Vector geometric approaches

Vector geometric approaches free us from the need to use coordinate systems as intermediaries when performing geometric operations or querying the relationship between objects. Instead they let us derive relevant expressions independent of any particular coordinate system.

Two primary considerations with respect to typical geometric analysis make the ability to define and manipulate expressions in this manner desirable:

- In general, no special relationships exist between the entities with respect to a given coordinate system that we can exploit to make the derivation easier.
- While in general no special relationship exists that we can exploit, there may coincidentally exist some special relationship that a coordinate-based algorithm would have to detect and handle.

Let's examine this second point more closely by again considering the example of rotating **u** onto **v**. Suppose we just consider the first part of that process, namely computing the matrix that rotates **u** onto the z-axis. Following the development in Foley et al.,[10] the required matrix would be the product

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \dfrac{\sqrt{u_x^2+u_z^2}}{|\mathbf{u}|} & \dfrac{-u_y}{|\mathbf{u}|} \\ 0 & \dfrac{u_y}{|\mathbf{u}|} & \dfrac{\sqrt{u_x^2+u_z^2}}{|\mathbf{u}|} \end{pmatrix} \begin{pmatrix} \dfrac{u_x}{\sqrt{u_x^2+u_z^2}} & 0 & \dfrac{u_z}{\sqrt{u_x^2+u_z^2}} \\ 0 & 1 & 0 \\ \dfrac{-u_z}{\sqrt{u_x^2+u_z^2}} & 0 & \dfrac{u_x}{\sqrt{u_x^2+u_z^2}} \end{pmatrix}$$

The difficulty arises because the denominators in the second matrix may be zero or nearly zero. Geometrically, this means that **u** lies parallel to (or nearly parallel to) the positive or negative y-axis direction. Therefore, an implementation of this coordinate-based algorithm would have to first check for this special case rather than simply computing the product of the two matrices. This proves unsatisfactory because whether **u** lies parallel to some coordinate axis is totally irrelevant to the operation ultimately desired, namely rotating **u** onto **v**. When computing the analogous two matrices for **v**, similar problems of course arise. Finally, the matrix for rotating **u** onto **v** is computed as the product of four matrices: the two matrices shown above and the inverse of the two **v** matrices.

By contrast, a vector geometric approach to this operation would simply compute directly the matrix that

rotates about the vector **w** (where $\mathbf{w} = \mathbf{u} \times \mathbf{v}$) by the acute angle between **u** and **v**. (Refer again to Figure 3.) We can easily determine the sine and cosine of this angle from the formulas presented in the section "Common vector operations." In Part 2 of this tutorial[1] we'll see a formula for the desired matrix that requires only **w**, the sine, and the cosine.

Having derived appropriate formulas, we must of course use some coordinate system so that we can generate internal numerical representations to drive our computer programs. The choice of coordinate system becomes irrelevant at that point—we'll get the same answer regardless of the coordinate system chosen.

Summary

After reviewing the basic operations in common interactive graphics pipelines, we discussed two different styles of analysis: vector geometric and coordinate-based. The former prove most applicable in situations where we can make no assumptions about how geometry relates to coordinate systems. Not only do our vector geometric expressions not rely upon special configurations of the geometry with respect to each other or to the coordinate system, they also don't require special-case handling when such special relationships occur.

As evidenced by these observations, mathematical derivations based on vector geometric analyses oftentimes prove simpler, stay free of annoying coordinate-system-dependent special cases, and produce results never worse and almost always better in terms of efficiency of implementation. Consequently, we see very real and quantifiable differences in the resulting code:

- more compact
- more computationally efficient
- more robust
- fewer, if any, special cases

Most special cases in vector geometric approaches are trivially detected before the algorithm begins. By contrast, coordinate-based methods commonly require special-case detection at intermediate steps of the algorithm based on quantities derived after a series of numerical computations. General rotations (or the example of rotating an arbitrary vector **u** onto another arbitrary vector **v**) offer excellent examples of this.

However, in some situations it's useful to transform geometry into special coordinate systems where we can exploit the system's orientation to dramatically simplify and accelerate certain types of imaging operations. Examples include clipping algorithms, visible line and visible surface determination, and intensity depth cueing.

Part 2 of this tutorial presents specific algorithms and techniques derived using these ideas.[1] I'll also show sample C++ code written using tools that implement the basic vector geometric operations discussed here. ■

Acknowledgments

Ron Goldman first introduced me to the power and simplicity of vector geometric approaches while we were working together in the early 1980s. I thank him for that as well as for his comments on an earlier draft of this material. The development of the software implementing the various techniques described here was performed in DesignLab, a multidisciplinary research laboratory at the University of Kansas funded in part by NSF Grant CDA-94-01021. The reviewers also provided valuable suggestions.

References

1. J.R. Miller, Applications of "Vector Geometry for Robustness and Speed," in preparation for *IEEE CG&A*, Vol. 19, No. 4, July/Aug. 1999.
2. J.R. Miller, "The Mathematics of Graphical Transformations: Vector Geometric and Coordinate-Based Approaches," DesignLab Tech. Report DL-1997-03, Univ. of Kansas, Lawrence, Ks., Jan. 1997.
3. T. DeRose, "A Coordinate-Free Approach to Geometric Programming," *Contemporary Approaches to Geometry for Computer Graphics and Computer-Aided Design*, Siggraph 89 Short Course No. 14, ACM Press, New York, Aug. 1989. (A condensed version also appeared in *Theory and Practice of Geometric Modeling*, W. Strasser and H. Seidel, eds., Springer Verlag, Berlin, 1989, pp. 291-306.)
4. R.N. Goldman, "Illicit Expressions in Vector Algebra," *ACM Trans. on Graphics*, Vol. 4, No. 3, July 1985, pp. 223-243.
5. R.N. Goldman, "Vector Geometry: A Coordinate-Free Approach," *Siggraph 85 Short Course No. 16*, ACM Press, New York, July 1985.
6. E. Abbott, *Flatland: A Romance of Many Dimensions*, Princeton University Press, Cambridge, Mass., 1991. (Originally published circa 1880.)
7. D. Burger, *Sphereland: A Fantasy about Curved Spaces and an Expanding Universe*, HarperPerennial, Boulder, Colo., 1965.
8. A.K. Dewdney, *The Planiverse: Computer Contact with a Two-Dimensional World*, Poseidon Books/Simon & Schuster, New York, 1984.
9. G. Farin, *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*, 4th edition, Academic Press, San Diego, 1996.
10. J.D. Foley et al., *Computer Graphics: Principles and Practice*, Addison-Wesley, Reading, Mass., 1990.

James R. Miller *is an associate professor of computer science in the Department of Electrical Engineering and Computer Science at the University of Kansas. His current research interests include computer graphics; geometric modeling; visualization; the use of computers and technology in education; and programming paradigms, especially object-oriented techniques and their applicability to problems in graphics, modeling, and visualization. He earned a BS in computer science from Iowa State University, and holds MS and PhD degrees in computer science from Purdue University.*

Readers may contact Miller at Dept. of Electrical Engineering and Computer Science, University of Kansas, 415 Snow Hall, Lawrence, KS 66045-2228, e-mail miller@eecs.ukans.edu.