Coordinate-Free Geometric Programming

Tony D. DeRose
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195

Technical Report 89-09-16
September 1989

ABSTRACT

These notes develop an algebra for geometric reasoning that is amenable to software implementation. The features of the algebra are chosen to support geometric programming of the variety found in computer graphics and computer aided geometric design applications. The implementation of the algebra in standard programming languages is described by defining an associated abstract data type.

The algebra and the abstract data type are founded on two basic principles: affine/Euclidean geometry and coordinate-freedom. Since many practitioners in graphics and computer aided geometric design have not studied these geometries in some time, a large part of the notes are devoted to a (relatively thorough) tutorial development that promotes geometric reasoning rather than coordinate calculations.

This work was supported in part by the National Science Foundation under grant numbers DMC-8602141 and DMC-8802949 and the Digital Equipment Corporation.

1. Introduction and Motivation

Computer graphics and computer aided geometric design (CAGD) applications are traditionally constructed by using a matrix package that implements homogeneous coordinates. Although this practice is widespread and successful, it does have its shortcomings. The basic problem with the traditional coordinate-based approach is revealed once it is recognized that there is a difference between matrix computations and geometric reasoning. Although graphics programs require geometric reasoning, traditional approaches offer only matrix computations; the geometric interpretation of these calculations is left to the imagination and discipline of the programmer.

Unfortunately, a given matrix computation can have many geometric interpretations. For instance, the code fragment shown in Figure 1 can be geometrically interpreted in at least three ways: as a change of coordinates, as a transformation from the plane onto itself, and as a transformation from one plane onto another (see Figure 2). The interpretation as a change of coordinates leaves the point unchanged geometrically, but changes the reference coordinate system (Figure 2(a)). The interpretation as a transformation of the plane onto itself moves the point, keeping the coordinate system fixed (Figure 2(b)). Finally, the interpretation as a transformation from one plane onto another involves two coordinate systems, one in the domain, and one in the range (Figure 2(c)).

$$\mathbf{P} \leftarrow \begin{pmatrix} p_1 & p_2 \end{pmatrix};$$
$$\mathbf{T} \leftarrow \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix};$$
$$\mathbf{P}' \leftarrow \mathbf{P}\,\mathbf{T};$$

FIGURE 1. A typical matrix computation.

A common response to this ambiguity is that it does not matter which view is taken. Indeed, this is the response that most students of computer graphics come to believe:

> "We have been discussing transforming a set of points belonging to an object into another set of points, with both sets in the same coordinate system. Thus the coordinate system stays unaltered and the object is transformed with respect to the origin of the coordinate system to obtain proper size. An alternate but equivalent way of thinking of a transformation is as a change of coordinate systems."
>
> —[Foley and Van Dam '82], p. 262

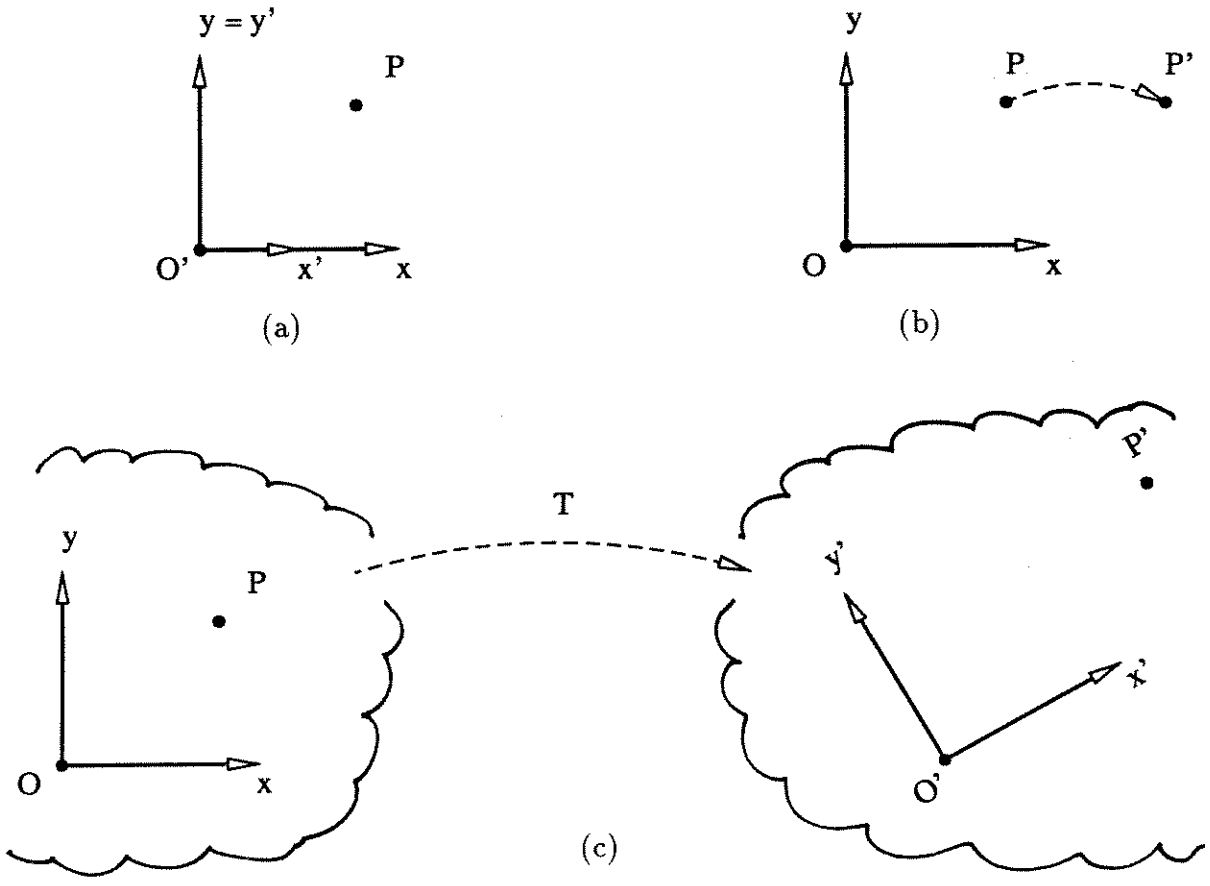

FIGURE 2. Three interpretations of the code fragment.

Unfortunately, this is not quite correct since it is possible to distinguish between the interpretations. In particular, lengths and angles don't change in the first interpretation, but they can in the second and third interpretations.

Above it was argued that a matrix computation could have many geometric interpretations. It is also the case that a matrix computation can have no geometric interpretation. As a particularly egregious example, suppose that P is a point in the projective plane having homogeneous coordinates $[1, 0, 1]$ (cf. [Foley and Van Dam '82][Riesenfeld '82]). The rules of homogeneous coordinates state that multiplying through by any non-zero factor does not alter the point being represented. Thus, P is also represented by the homogeneous coordinates $[-1, 0, -1]$. Now, suppose we add these two matrix representations together:

$$[1, 0, 1] + [-1, 0, -1] = [0, 0, 0],$$

which is certainly valid as a matrix computation. The problem is that $[0, 0, 0]$ is not a valid homogeneous coordinate (cf. [Riesenfeld '82]). In other words, there is no point with

homogeneous coordinates $[0, 0, 0]$, implying that the matrix computation we performed is not geometrically valid.

Although the previous example is particularly nasty, there are more common errors that can occur when using traditional coordinate-based approaches. Some errors are allowed to creep in because distinct coordinate systems are not explicitly represented. The applications programmer is expected to maintain a clear idea of which coordinate system (e.g., world coordinates, viewing coordinates, etc.) each point is represented in. As a consequence, the burden of coordinate transformations must be borne by the programmer. If extreme care is not taken, it is possible (and in fact common) to perform geometrically meaningless operations such as combining two points that are represented relative to different coordinate systems.

Fortunately, the problems of ambiguity and validity can be solved. The solution presented here consists of two subtasks. The first task is the identification of a *geometric algebra* (i.e., a collection of geometric objects and operations for manipulating them) that provides concepts and computational tools necessary for typical graphics and CAGD applications. The second task is to implement this algebra as an abstract data type (ADT) that presents the underlying geometric concepts in a clean, clear fashion. The algebra and the ADT are, of necessity, very closely related. In particular, the collection of objects in the algebra become the types of the ADT, and the operations in the algebra become the procedures of the ADT. Although the ADT is ultimately implemented as a collection of coordinated matrix computations, the abstraction presented to the programmer is a purely geometric one. The algebra and ADT are constructed so that only geometrically meaningful operations are possible. Moreover, all operations are geometrically unambiguous and their interpretation is clearly reflected in the code.

At the heart of the algebra and the ADT are two central themes: affine/Euclidean geometry and coordinate-freedom. The use of the theory of affine/Euclidean geometry facilitates the identification of useful geometric entities and operations; the use of coordinate-freedom forces the programmer to use geometric operations rather than coordinate calculations. The ADT is then given the responsibility of translating the geometric operations into coordinate calculations at a level beneath the programmer.

Unfortunately, abstraction does not come without expense. In this instance, the expense is three-fold:

1. The coordinate-free approach to affine/Euclidean geometry must be learned by applications programmers.
2. The ADT we are proposing is more difficult to implement than standard matrix packages. In fact, the ADT is most conveniently built on top of a matrix package.
3. A run-time performance penalty is incurred. There are two sources of degraded performance. The first occurs due to run-time type checking. Actually, this is only

a problem if the ADT is implemented in standard languages such as C, Modula 2, and Ada. If the ADT is implemented by defining a language and compiler, then most of the required type-checking can be done at compile-time. The second source of performance degradation is a result of coordinate-freedom. In the coordinate-free approach, the programmer does not have control over the coordinate system in which the geometric objects are represented. Consequently, geometric operations may not be performed as efficiently as otherwise possible. As a specific example taken from ray tracing, if complete control over coordinates is possessed by the programmer (as is typically the case), then the test to determine if a line segment crosses the $x = 0$ plane can be accomplished simply by checking the signs of the x-components of the endpoints. In contrast, if the programmer uses the coordinate-free ADT, this test requires six multiplications and eight additions.

The important trade-off then is between ease of development, debugging, and maintenance on one hand, and raw performance on the other. This is, of course, one of the classic trade-offs in software development. It occurs, for instance, any time a high-level language is used in place of assembly language.

The remainder of these notes is divided into two major parts. The first part, consisting of Sections 2 through 4, presents a tutorial introduction to affine and Euclidean geometry. The second part, consisting of Sections 5, 6, and Appendix 2, describes the abstract data type, examples of its use, and details of its implementation.

Although the development of affine and Euclidean geometry is done in a coordinate-free way, the geometric ADT is implemented using coordinates. It is therefore important for the implementor of the ADT to understand the how to translate geometric operations into coordinate calculations. In an effort to clearly separate the coordinate-free material from the coordinate-dependent material, the coordinate-dependent sections have been marked with an asterisk.

2. The Search for a Geometric Algebra

To address the first task, that of identifying an appropriate geometric algebra, we note that the theory should provide a model of the geometric spaces one encounters in typical graphics and CAGD applications, suggesting that Euclidean geometry may be the correct choice. In fact, Euclidean geometry (and the closely related theory of affine geometry) will be the one presented here. However, before developing that theory and the ADT based on it, it is worthwhile discussing two other candidate theories; namely, the theory of vector spaces (linear algebra), and the theory of projective spaces (projective geometry).

The connection between linear algebra and geometry is a common topic in elementary linear algebra courses. The theory successfully models some aspect of geometry in that it can be used to derive matrices that describe rotation, scaling, and skew. The theory

of vector spaces has an unfortunately subtle shortcoming: in every vector space there is a distinguished origin or zero vector, whereas in the geometric spaces encountered in graphics and CAGD, there is no absolute origin. This seemingly minor difference has striking undesirable implications for graphics and CAGD applications, as has been detailed by Goldman [Goldman '85][Goldman '87] and others. Thus, vector spaces do not provide an appropriate abstraction for these endeavors.

We turn now to projective geometry. Historically, projective geometry was invented to eliminate certain nagging degeneracies that are encountered in Euclidean spaces. For example, in the Euclidean plane nearly every pair of distinct lines intersect. However, some lines are parallel and hence do not intersect. In projective geometry the situation is much more uniform. In the projective plane for instance, every distinct pair of lines is assumed to intersect in exactly one point. This uniformity creates certain topological differences between the Euclidean plane and the projective plane, as is nicely pointed out in [Blinn and Newell '81] and in [Riesenfeld '81].

For our purposes, the major ramification of the change in topology is that the notion of parallelism is forfeit in projective geometry (there is no way to define it since all lines intersect). This is dreadful in the context of graphics and CAGD applications since the notion of parallelism is absolutely essential. In fact, without parallelism it is not possible to define "free vectors," the kind of computationally useful vectors one encounters in vector analysis and physics. If vectors and projective geometry (or equivalently, homogeneous coordinates) are truly antithetical, then something must clearly be amiss because it is common practice in graphics to intermingle these ideas within the same program. The mixing of affine and projective concepts can obviously be made to work, as demonstrated by the fact that these programs generally produce the expected results. However, even if one does not adopt the purely affine/Euclidean viewpoint espoused here, it is important to recognize and understand how affine and projective ideas can be intertwined to produce "correct" results; to this end we offer Section 3.4.

3. Affine Geometry

Although the geometric ADT will ultimately be based on Euclidean geometry, many of the geometric objects and operations that find use in computer graphics and CAGD are founded in the more general field of affine geometry. We have therefore chosen to develop the more general theory here, then specialize to Euclidean geometry in Section 4.

There are many different approaches to affine geometry [Weyl '22] [Kowalsky '71][Flohr and Raith '74][Dodson and Poston '79]. One approach, first put forth by Weyl [Weyl '22] (a modern account of which can be found in [Dodson and Poston '79] and [Flohr and Raith '74]), makes a distinction between points and vectors, but does not define operations for combining them. The method we shall adopt is very similar to that used by Dodson and

Poston. This development of affine geometry builds on vector spaces, so a brief review of the relevant parts of linear algebra is supplied in Appendix 1.

3.1. Affine Spaces

The most basic objects in our geometric algebra will be *affine spaces*, which in turn consist of *points* and *free vectors*. Intuitively, the only thing that distinguishes one point from another is its position. In more computer-sciencey jargon, points only have a position attribute. Free vectors on the other hand have the attributes of magnitude and direction, but no fixed position; the modifier "free" therefore refers to the ability of vectors to move about in the space. Free vectors will henceforth be referred to simply as vectors.

Geometrically we draw points such as P and Q as dots, and we draw vectors such as $\vec{v}$ and $\vec{w}$ as line segments with arrow heads (see Figure 3). (To avoid confusion about which symbols are points and which are vectors, we will conform to the convention that points will be written in upper case and vectors will be written in lower case and will be ornamented with a diacritical arrow.)

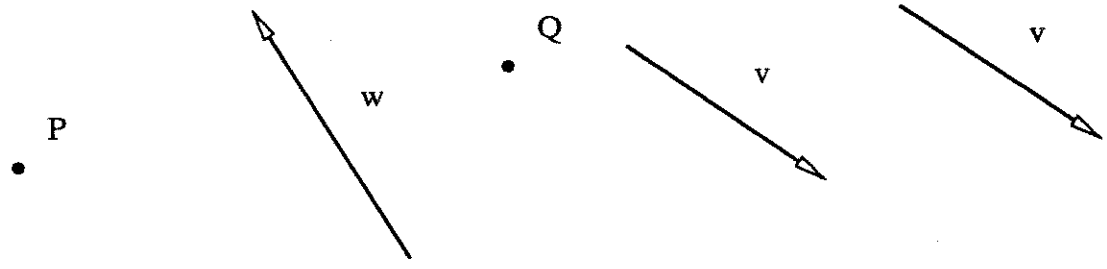

FIGURE 3. Geometric interpretations of points and vectors.

More formally, an affine space $\mathcal{A}$ is a pair $(\mathcal{P}, \mathcal{V})$ where $\mathcal{P}$ is the set of points and $\mathcal{V}$ is the set of vectors. We shall use the notation $\mathcal{A}.\mathcal{P}$ and $\mathcal{A}.\mathcal{V}$ to refer to the points and vectors of an affine space $\mathcal{A}$. The vectors of an affine space are assumed to form a vector space. If n denotes the dimension of the vector space, then the affine space is called an affine n-space. An affine 1-space is more commonly called an *affine line*, and an affine 2-space is more commonly called an *affine plane*.

The set of points and the vector space of an affine space $\mathcal{A}$ are related through the following axioms:

(i) *Subtraction*: There exists an operation of subtraction that satisfies:

a. For every pair of points P, Q, there is a unique vector $\vec{v}$ such that $\vec{v} = P - Q$.

b. For every point Q and every vector $\vec{v}$, there is a unique point P such that $P - Q = \vec{v}$.

(ii) *The Head-to-Tail Axiom*: Every triple of points P, Q and R, satisfies

$$(P - Q) + (Q - R) = P - R.$$

Before describing in more detail what the axioms mean geometrically, it is convenient to use the them to define the operation of addition between points and vectors. Specifically, we define $Q + \vec{v}$ to be the unique point P such that $P - Q = \vec{v}$. The geometric interpretation of addition is shown in Figure 4.

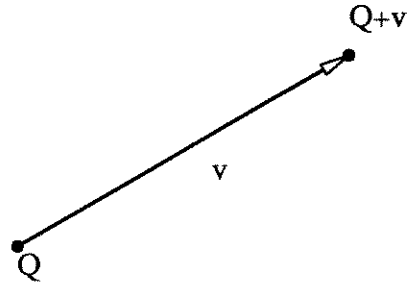

FIGURE 4. Addition of points and vectors.

In terms of the addition operation, axiom (i) states that if point Q is fixed, then there is a one-to-one correspondence between vectors $(\vec{v})$ and points $(Q + \vec{v})$. The vector connecting the points Q and P can therefore be labeled as $P - Q$, as shown in Figure 5.

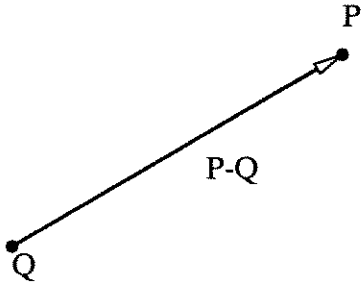

FIGURE 5. Subtraction of points.

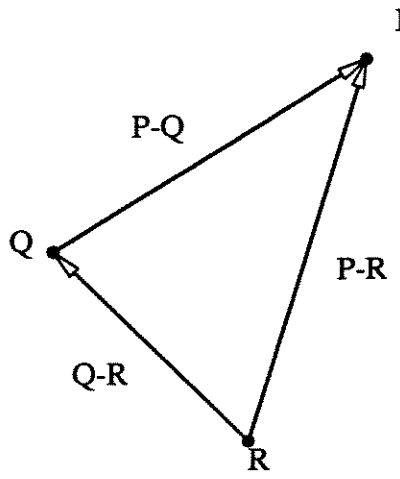

FIGURE 6. The head-to-tail axiom.

The geometric interpretation of axiom (ii), shown in Figure 6, indicates that the axiom is actually a statement of the familiar "head-to-tail rule" for vector addition, stated in terms of points rather than vectors. (Recall from elementary vector analysis that the vector addition $\vec{v} + \vec{w}$ can be constructed geometrically by aligning the head of $\vec{v}$ with the tail of $\vec{w}$. The sum is then the vector from the tail of $\vec{v}$ to the head of $\vec{w}$.)

EXAMPLE 3.1: Examples of affine spaces abound. For instance, if you believe that time is infinite, then the time line is an example of a (one dimensional) affine space. The points of the affine space correspond to dates, and the vectors of the affine space correspond to numbers of days. A date (a point) minus another date is a number of days (a vector). Thus, subtraction of points makes sense as a vector. The other axioms can also be shown to hold.

The theory of polynomials can be used as the source of another example of an affine space. Let the set of vectors be the set of homogeneous cubic polynomials (a polynomial is said to be homogeneous if its constant coefficient is zero). The set of points can then be taken to be the set of cubic polynomials whose constant term is 1. It is a simple matter to show that the axioms hold when standard polynomial addition and subtraction are used to add points to vectors and to subtract points. The dimension of this affine space is 3 since that is the dimension of the space of homogeneous cubic polynomials. □

Several simple deductions can be made from the head-to-tail axiom. By setting $Q = R$, we find that $(P-Q)+(Q-Q) = P-Q$, which implies that $Q-Q$ must be the zero vector $\vec{0}$ since adding it to $P-Q$ results in $P-Q$. By setting $P = R$, we see that $(R-Q)+(Q-R) = \vec{0}$, implying that $R - Q = -(Q - R)$. These facts, along with several others, are summarized in the following claim.

CLAIM 3.1: The following identities hold for all points P, Q and R, and all vectors $\vec{v}$ and $\vec{w}$.

(a) $Q - Q = \vec{0}$.
(b) $R - Q = -(Q - R)$.
(c) $\vec{v} + (Q - R) = (Q + \vec{v}) - R$.
(d) $Q - (R + \vec{v}) = (Q - R) - \vec{v}$.
(e) $P = Q + (P - Q)$.
(f) $(Q + \vec{v}) - (R + \vec{w}) = (Q - R) + (\vec{v} - \vec{w})$.

PROOF: Parts (a) and (b) were proved above. To prove (c), let point P be defined by $\vec{v} = P - Q$. The head-to-tail axiom then says that $\vec{v} + (Q - R) = P - R$. The proof is completed by substituting $Q + \vec{v}$ for P. Part (d) follows immediately from (c) by multiplying through by -1. To prove (e), use the definition of addition together with the head-to-tail axiom to write P in the form $P = R + (P - Q) + (Q - R)$. Now, taking $Q = R$ we find that $P = Q + (P - Q) + \vec{0}$, which completes the proof since adding the zero vector to the right side has no affect.

The proof of (f) is somewhat harder as it requires the two invocations of the head-to-tail axiom together with the use of parts (a), (c) and (d):

$$\begin{aligned}
(Q + \vec{v}) - (R + \vec{w}) &= [(Q + \vec{v}) - R] + [R - (R + \vec{w})] && \text{– by head-to-tail axiom} \\
&= [(Q + \vec{v}) - R] + [(R - R) - \vec{w}] && \text{– by part (d)} \\
&= [(Q + \vec{v}) - R] - \vec{w} && \text{– by part (a)} \\
&= [(Q + \vec{v}) - Q] + [Q - R] - \vec{w} && \text{– by head-to-tail axiom} \\
&= [\vec{v} + (Q - Q)] + [Q - R] - \vec{w} && \text{– by part (c)} \\
&= (Q - R) + (\vec{v} - \vec{w}) && \text{– by part (a)}
\end{aligned}$$

□

Thus far, the operations in the algebra can be summarized as:

$$\begin{aligned}
\text{vector} + \text{vector} &\mapsto \text{vector} \\
\text{vector} * \text{scalar} &\mapsto \text{vector} \\
\text{point} - \text{point} &\mapsto \text{vector} \\
\text{point} + \text{vector} &\mapsto \text{point}.
\end{aligned}$$

Notice the asymmetry in the way points and vectors are handled. In particular, notice that it is possible to add vectors, but addition of points is not defined. Similarly, the process of multiplying a point by a scalar is undefined. The asymmetry shouldn't be too surprising since points and vectors are being used in very different ways. In some respects the points are the primary objects of the geometry, whereas the role of the vectors is to allow movement from point to point by employing the operation of addition between points and vectors. In Section 3.2, we will see that the vectors are also used to introduce coordinates.

Although we may not be able to add two points, there are certain other convenient operations that can be defined. For instance, consider the expression

$$Q = Q_1 + \alpha(Q_2 - Q_1), \tag{3.1}$$

where Q_1, Q_2 are points and α is a scalar. This expression is meaningful in the context of our algebra because $Q_2 - Q_1$ is meaningful as a vector, implying that $\alpha(Q_2 - Q_1)$ is meaningful as a vector, implying that Q is meaningful as a point since it is the result of adding a point and a vector. Geometrically this means that point Q is one α^{th} along the way from the point Q_1 to the point Q_2, as shown in Figure 7.

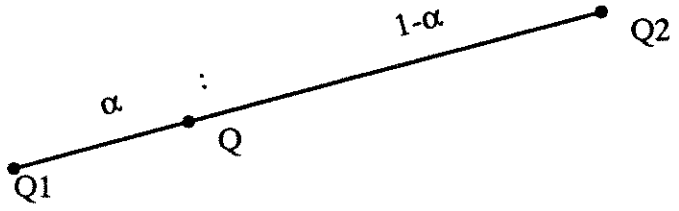

FIGURE 7. Geometric interpretation of Equation (3.1).

If we forget for a moment that we are dealing with points, vectors, and scalars, we might be tempted to algebraically rearrange Equation (3.1) into the form

$$Q = (1 - \alpha)Q_1 + \alpha Q_2,$$

or perhaps in the more symmetric form

$$Q = \alpha_1 Q_1 + \alpha_2 Q_2, \qquad \alpha_1 + \alpha_2 = 1. \tag{3.2}$$

This equation looks a bit odd since it appears that we are multiplying points by scalars (an undefined operation), then adding the result together (also undefined). We can formally get out of this bind by making a new definition.

Definition: The expression

$$\alpha_1 Q_1 + \alpha_2 Q_2, \tag{3.3}$$

where $\alpha_1 + \alpha_2 = 1$ is defined to be the point

$$Q_1 + \alpha_2(Q_2 - Q_1).$$

An expression such as Equation (3.3) is called an *affine combination.* Affine combinations possess simple geometric interpretations. In particular, Equation (3.3) states that the point Q lies on the line segment Q_1, Q_2 so as to break the segment into relative distances $\alpha_2 : \alpha_1$,

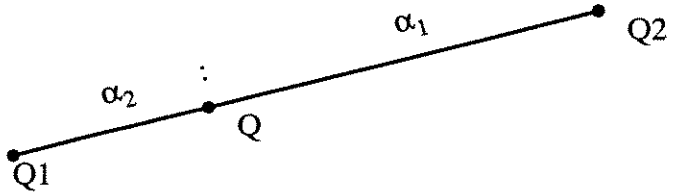

FIGURE 8. Point Q breaks Q_1Q_2 into relative ratios $\alpha_2 : \alpha_1$.

as shown in Figure 8. Conversely, if a point Q is known to break a line segment Q_1, Q_2 into relative ratios $a : b$, then Q can be expressed as

$$Q = \frac{bQ_1 + aQ_2}{a + b},$$

where for the sake of generality we have not assumed that a and b sum to one.

The notion of an affine combination can be generalized to allow the combination of an arbitrary number of points. If $Q_1, ..., Q_k$ are points and $\alpha_1, ..., \alpha_k$ are real numbers that sum to unity, then

$$\alpha_1 Q_1 + \alpha_2 Q_2 + \alpha_3 Q_3 + \cdots + \alpha_k Q_k$$

is defined to be the point

$$Q_1 + \alpha_2(Q_2 - Q_1) + \alpha_3(Q_3 - Q_1) + \cdots + \alpha_k(Q_k - Q_1).$$

REMARK: As an aside of interest only to the purist, we mention another approach to affine geometry. An affine space can be defined as a set S that is closed under affine combinations. The points of the affine space are the elements of S; the vectors are then defined to be equivalence classes of ordered pairs of points. The equivalence relation is constructed to build in the head-to-tail axiom. In particular, two pairs of points (Q_1, P_1) and (Q_2, P_2) are said to be equivalent if

$$\frac{Q_1 + P_2}{2} = \frac{Q_2 + P_1}{2}.$$

This condition has the geometric interpretation that $Q_1P_1P_2Q_2$ must form a parallelogram, as shown in Figure 9. It is not too hard to show that this condition is an equivalence relation on the ordered pairs of points, implying that the set of all ordered pairs of points are partitioned into equivalence classes. If $[Q, P]$ denotes the equivalence class containing the pair (Q, P), then the set of all equivalence classes form a vector space, with scalar multiplication and addition defined as:

- $\alpha[Q, P] = [Q, (1 - \alpha)Q + \alpha P], \alpha \in \Re$;
- $[Q_1, P_1] + [Q_2, P_2] = [Q_1, P_1 + P_2 - Q_2]$.

The elements of the vector space thus formed are the vectors of the affine space.

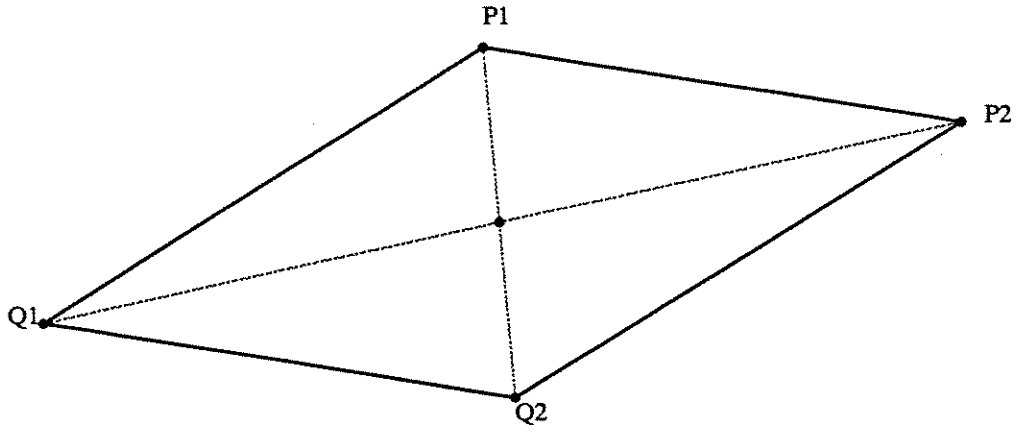

FIGURE 9. Points $Q_1P_1P_2Q_2$ forming a parallelogram.

3.2. Frames

To perform numerical computations and to facilitate the conversion between coordinates and geometric entities, we must understand how affine spaces are coordinatized. In this section, we give two methods for imposing coordinates on affine spaces: frames and simplexes. Each method has its advantages, but we have chosen to use frames in the ADT since they are more familiar to those used to traditional approaches to geometric programming.

Let $\mathcal{A} = (\mathcal{P}, \mathcal{V})$ be an affine n-space, let $\mathcal{O}$ be any point, and let $\vec{v}_1, \cdots, \vec{v}_n$ be any basis for $\mathcal{A}.\mathcal{V}$. We call the collection $(\vec{v}_1, \cdots, \vec{v}_n, \mathcal{O})$ a *frame* for $\mathcal{A}$. Frames play the same role in affine geometry that bases play in vector spaces. The role of frames is more precisely indicated by the next claim.

CLAIM 3.2: If $\mathcal{F} = (\vec{v}_1, ..., \vec{v}_n, \mathcal{O})$ is a frame for some affine n-space, then every vector $\vec{u}$ can be written uniquely as

$$\vec{u} = u_1\vec{v}_1 + u_2\vec{v}_2 + \cdots + u_n\vec{v}_n, \tag{3.4}$$

and every point P can be written uniquely as

$$P = p_1\vec{v}_1 + p_2\vec{v}_2 + \cdots + p_n\vec{v}_n + \mathcal{O}. \tag{3.5}$$

The sets of scalars $(u_1, u_2, ..., u_n)$ and $(p_1, p_2, ..., p_n)$ are called the *affine coordinates of* $\vec{u}$ *and* P *relative to* $\mathcal{F}$.

PROOF: The unique representation of $\vec{u}$ follows from the fact that $(\vec{v}_1, ..., \vec{v}_n)$ forms a basis for $\mathcal{A}.\mathcal{V}$. From the definition of addition between points and vectors, there is a unique vector $\vec{w}$ such that

$$P = \vec{w} + \mathcal{O}.$$

Since $(\vec{v}_1, ..., \vec{v}_n)$ is a basis for $\mathcal{A}.\mathcal{V}$, $\vec{w}$ has a unique representation

$$\vec{w} = p_1\vec{v}_1 + p_2\vec{v}_2 + \cdots + p_n\vec{v}_n.$$

Thus, P can be expressed uniquely as

$$P = p_1\vec{v}_1 + p_2\vec{v}_2 + \cdots + p_n\vec{v}_n + \mathcal{O}.$$

□

Equation (3.5) can be written in a more symmetric form as an affine combination of $n+1$ points. Let $Q_i = \mathcal{O} + \vec{v}_i$ for $i = 1, ..., n$, set Q_0 equal to $\mathcal{O}$, and let $p_0 = 1 - (p_1 + \cdots + p_n)$. With these definitions, simple rearrangement allows Equation (3.5) to be rewritten as

$$P = p_0Q_0 + p_1Q_1 + \cdots + p_nQ_n, \tag{3.6}$$

where, by construction, $p_0 + p_1 + \cdots + p_n = 1$. Since every point can be written uniquely in the form of Equation (3.5), every point can also be written uniquely in the form of (3.6). In this form, the scalars $(p_0, ..., p_n)$ are called the *barycentric coordinates* of P relative to the *n-simplex* $Q_0, ..., Q_n$. An n-simplex is a collection of $n+1$ points such that none of the points can be expressed as an affine combination of the others. Thus, a 1-simplex is a line segment, a 2-simplex is a triangle, a 3-simplex is a tetrahedron, and so forth.

Vectors can also be represented in barycentric form as follows. By letting $u_0 = -(u_1 + \cdots + u_n)$, Equation (3.4) can be rewritten as

$$\vec{u} = u_0Q_0 + u_1Q_1 + \cdots + u_nQ_n,$$

where, by construction, $u_0 + u_1 + \cdots + u_n = 0$.

Simplexes and barycentric coordinates therefore offer an alternative method of introducing coordinates into an affine space. If the coordinates sum to one, they represent a point; if the coordinates sum to zero, they represent a vector. The notion of barycentric coordinates may at first seem somewhat obscure, but it is actually used in several situations in graphics and CAGD. For instance, one can view the Gouraud shading of triangles as an application of barycentric coordinates. Recall that in Gouraud shading each point P of a triangle is assigned a color that is a linear blend of the colors at the vertices Q_0, Q_1 and Q_2. In equation form,

$$\text{Color}(P) = p_0\text{Color}(Q_0) + p_1\text{Color}(Q_1) + p_2\text{Color}(Q_2),$$

where p_0, p_1 and p_2 are weighting factors that vary linearly over the face of the triangle. These weighting factors are in fact the barycentric coordinates of P relative to Q_0, Q_1, Q_2, as can be shown by using results from Section 3.3.

Simplexes and barycentric coordinates also have important uses in the theory of Bézier curves and surfaces (cf. [de Boor '87][Farin '86]).

* 3.2.1. Matrix Representations of Points and Vectors

In the previous section it was shown that points and vectors can be uniquely identified by their coordinates relative to a given frame. The most straightforward way to represent points and vectors in a computer is then to simply store their coordinates as a $1 \times n$ row matrix. However, for reasons that will only become fully apparent later, it is more convenient to augment the row matrix with an additional value that distinguishes between points and vectors [Goldman '87]. To allow this augmentation to proceed in a rigorous fashion, we extend the original set of axioms for an affine space $\mathcal{A}$ to include:

(iii) *Coordinate Axiom:* For every point $P \in \mathcal{A}.\mathcal{P}$, $0 \cdot P = \vec{0}$, the zero vector of $\mathcal{A}.\mathcal{V}$, and $1 \cdot P = P$.

Armed with this axiom, we can rewrite Equation (3.5) in matrix notation as

$$\begin{aligned} P &= p_1\vec{v}_1 + p_2\vec{v}_2 + \cdots + p_n\vec{v}_n + 1 \cdot \mathcal{O} \\ &\quad (\, p_1 \quad p_2 \quad \cdots \quad p_n \quad 1\,)(\, \vec{v}_1 \quad \vec{v}_2 \quad \cdots \quad \vec{v}_n \quad \mathcal{O}\,)^T. \end{aligned}$$

Notice that the last component in the row matrix essentially says that P is a point. Vectors can be represented in a similar fashion by rewriting Equation (3.4) as:

$$\begin{aligned} \vec{u} &= u_1\vec{v}_1 + u_2\vec{v}_2 \cdots u_n\vec{v}_n + 0 \cdot \mathcal{O} \\ &= (\, u_1 \quad u_2 \quad \cdots \quad u_n \quad 0\,)(\, \vec{v}_1 \quad \vec{v}_2 \quad \cdots \quad \vec{v}_n \quad \mathcal{O}\,)^T. \end{aligned}$$

Thus, vectors are represented as row matrices whose last component is zero.

REMARK: Those familiar with homogeneous coordinates are used to adding an additional component when representing points. Note however that it has been done here without having to mention homogeneous coordinates or projective spaces. This is not simply a trick, for we are representing affine entities, not projective ones. For instance, in the current context, a row matrix with a final component of zero represents a vector, whereas in projective geometry, a row matrix with a final component of zero represents an *ideal point* (more commonly known as a point at infinity). □

Suppose that a point P has coordinates $(p_1, ..., p_n, 1)$ relative to a frame $\mathcal{F} = (\vec{v}_1, ..., \vec{v}_n, \mathcal{O})$. It is natural to ask: what are the coordinates of P relative to a frame $\mathcal{F}' = (\vec{v}'_1, ..., \vec{v}'_n, \mathcal{O}')$? To answer this question, we must find scalar values $p'_1, ..., p'_n$ such that

$$p'_1\vec{v}'_1 + \cdots + p'_n\vec{v}'_n + \mathcal{O}' = p_1\vec{v}_1 + \cdots + p_n\vec{v}_n + \mathcal{O}.$$

It is more convenient to write this equation in matrix notation as

$$\begin{pmatrix} p'_1 & \cdots & p'_n & 1 \end{pmatrix} \begin{pmatrix} \vec{v}'_1 \\ \vdots \\ \vec{v}'_n \\ \mathcal{O}' \end{pmatrix} = \begin{pmatrix} p_1 & \cdots & p_n & 1 \end{pmatrix} \begin{pmatrix} \vec{v}_1 \\ \vdots \\ \vec{v}_n \\ \mathcal{O} \end{pmatrix}. \tag{3.7}$$

Each of the elements of $\mathcal{F}$ can be written in coordinates relative to $\mathcal{F}'$. In particular, let these coordinates be such that:

$$\vec{v}_i = f_{i,1}\vec{v}'_1 + \cdots + f_{i,n}\vec{v}'_n$$
$$\mathcal{O} = f_{n+1,1}\vec{v}'_1 + \cdots + f_{n+1,n}\vec{v}'_n + \mathcal{O}'$$

for $i = 1, ..., n$. Substituting these equations into Equation (3.7) gives

$$\begin{pmatrix} p'_1 & \cdots & p'_n & 1 \end{pmatrix} \begin{pmatrix} \vec{v}'_1 \\ \vdots \\ \vec{v}'_n \\ \mathcal{O}' \end{pmatrix} = \begin{pmatrix} p_1 & \cdots & p_n & 1 \end{pmatrix} \begin{pmatrix} f_{1,1}\vec{v}'_1 + \cdots + f_{1,n}\vec{v}'_n \\ \vdots \\ f_{n,1}\vec{v}'_1 + \cdots + f_{n,n}\vec{v}'_n \\ f_{n+1,1}\vec{v}'_1 + \cdots + f_{n+1,n}\vec{v}'_n + \mathcal{O}' \end{pmatrix},$$

which can be rewritten as

$$\begin{pmatrix} p'_1 & \cdots & p'_n & 1 \end{pmatrix} \begin{pmatrix} \vec{v}'_1 \\ \vdots \\ \vec{v}'_n \\ \mathcal{O}' \end{pmatrix} = \begin{pmatrix} p_1 & \cdots & p_n & 1 \end{pmatrix} \begin{pmatrix} f_{1,1} & \cdots & f_{1,n} & 0 \\ \vdots & \ddots & \vdots & \vdots \\ f_{n,1} & \cdots & f_{n,n} & 0 \\ f_{n+1,1} & \cdots & f_{n+1,n} & 1 \end{pmatrix} \begin{pmatrix} \vec{v}'_1 \\ \vdots \\ \vec{v}'_n \\ \mathcal{O}' \end{pmatrix}.$$

Linear independence of the vectors $\vec{v}'_1, ..., \vec{v}'_n$ can be used to deduce that

$$\begin{pmatrix} p'_1 & \cdots & p'_n & 1 \end{pmatrix} = \begin{pmatrix} p_1 & \cdots & p_n & 1 \end{pmatrix} \begin{pmatrix} f_{1,1} & \cdots & f_{1,n} & 0 \\ \vdots & \ddots & \vdots & \vdots \\ f_{n,1} & \cdots & f_{n,n} & 0 \\ f_{n+1,1} & \cdots & f_{n+1,n} & 1 \end{pmatrix}.$$

Thus, a change of coordinate systems can be accomplished via matrix multiplication. Notice that the matrix used to affect the change of coordinates has rows consisting of the coordinates of the elements of the "old frame" (frame $\mathcal{F}$) relative to the "new frame" (frame $\mathcal{F}'$).

3.3. Affine Transformations

The next geometric object to be added to our collection is the affine transformation. Affine transformations are mappings between affine spaces that preserve the algebraic structure of the spaces. That is, affine transformations map points to points, vectors to vectors, frames to frames, and so forth.

To begin, let $\mathcal{A}$ and $\mathcal{B}$ be two affine spaces (it is sometimes the case that $\mathcal{A}$ and $\mathcal{B}$ are the same space). A map $F : \mathcal{A}.\mathcal{P} \mapsto \mathcal{B}.\mathcal{P}$ is said to be an *affine transformation* (also called an affine map) if it preserves affine combinations. That is, F is an affine map if the condition

$$F(\alpha_1 Q_1 + \alpha_2 Q_2 + \cdots + \alpha_k Q_k) = \alpha_1 F(Q_1) + \alpha_2 F(Q_2) + \cdots + \alpha_k F(Q_k) \qquad (3.8)$$

holds for all points $Q_1, ..., Q_k$ and for all sets of α's that sum to unity. (Notice the similarity between this definition and the definition of linear transformations given in Appendix 1.) Examples of affine transformations include: reflections, skew transformations, translations, scalings, and orthogonal projections. However, perspective projections are not affine transformations, but they are *projective* transformations (see Section 3.4).

EXAMPLE 3.2: As a specific example of an affine transformation, consider the transformation $T : \mathcal{A}.\mathcal{P} \mapsto \mathcal{A}.\mathcal{P}$ that performs translation along a fixed vector $\vec{t}$. This transformation can be defined by

$$T(P) = P + \vec{t}.$$

To show that T is an affine transformation, it suffices to show that

$$T(\alpha_1 P_1 + \alpha_2 P_2) = \alpha_1 T(P_1) + \alpha_2 T(P_2)$$

for every pair of points P_1, P_2, and for every α_1, α_2 such that $\alpha_1 + \alpha_2 = 1$. This is not difficult to do, as the following derivation shows:

$$\begin{aligned}
T(\alpha_1 P_1 + \alpha_2 P_2) &= (\alpha_1 P_1 + \alpha_2 P_2) + \vec{t} && - \\
&= P_1 + \alpha_2(P_2 - P_1) + \vec{t} && - \text{ by def of affine comb} \\
&= (P_1 + \vec{t}) + \alpha_2[(P_2 - P_1) + (\vec{t} - \vec{t})] && - \\
&= (P_1 + \vec{t}) + \alpha_2[(P_2 + \vec{t}) - (P_1 + \vec{t})] && - \text{ by Claim 3.1(f)} \\
&= T(P_1) + \alpha_2(T(P_2) - T(P_1)) && - \\
&= \alpha_1 T(P_1) + \alpha_2 T(P_2) && - \text{ by def of affine comb}
\end{aligned}$$

Induction on the number of terms in the affine combination can be used to show that T preserves arbitrary affine combinations. □

An immediate consequence of their definition is that affine transformations carry line segments to line segments, and hence, planes to planes and hyperplanes to hyperplanes. This can be seen by noting that the line segment connecting the points Q_1 and Q_2 can be written in parametric form as the affine combination

$$Q(t) = (1 - t)Q_1 + tQ_2, \qquad t \in [0, 1]. \qquad (3.9)$$

The image of the segment under an affine map F is therefore

$$F(Q(t)) = (1 - t)F(Q_1) + tF(Q_2), \qquad t \in [0, 1], \qquad (3.10)$$

which is a parametric description of the line segment connecting the images of Q_1 and Q_2. Equations (3.9) and (3.10) actually show something substantially stronger. In particular, they show that the point breaking the line segment Q_1, Q_2 into relative ratio $t : (1-t)$ is mapped to the point that breaks $F(Q_1), F(Q_2)$ into the same relative ratio, as shown in Figure 10. We therefore arrive at the important fact that affine maps preserve relative ratios.

Another important fact about affine maps is that they are completely determined if the image of an n-simplex is known. To see this, let $F : \mathcal{A}.\mathcal{P} \mapsto \mathcal{B}.\mathcal{P}$ be an affine map, let $\mathcal{A}$ be an affine n-space, and let $Q_0, ..., Q_n$ be an n-simplex in $\mathcal{A}$. In the previous section it was shown that every point in $\mathcal{A}$ can be written uniquely as $P = p_0Q_0 + \cdots + p_nQ_n$. The fact that F is affine implies that $F(P) = p_0F(Q_0) + \cdots + p_nF(Q_n)$, which is completely determined if the image of the simplex $F(Q_0), ..., F(Q_n)$ is known.

We can push the above argument a bit further to yield another interesting result. Suppose that $S = (Q_0, ..., Q_n)$ and $S' = (Q'_0, ..., Q'_n)$ are arbitrary simplexes in $\mathcal{A}$ and $\mathcal{B}$, respectively. We claim that there is a unique map from $\mathcal{A}$ to $\mathcal{B}$ that carries S into S'. The proof is immediate: existence follows by letting the map F in the previous paragraph be such that $F(Q_i) = Q'_i$, for $i = 0, ..., n$; uniqueness follows from the fact that every point has a unique barycentric representation. In the case of affine planes, this result says that every pair of triangles are related by a unique affine map. Similarly, every pair of tetrahedrons in an affine 3-space are related by a unique affine map.

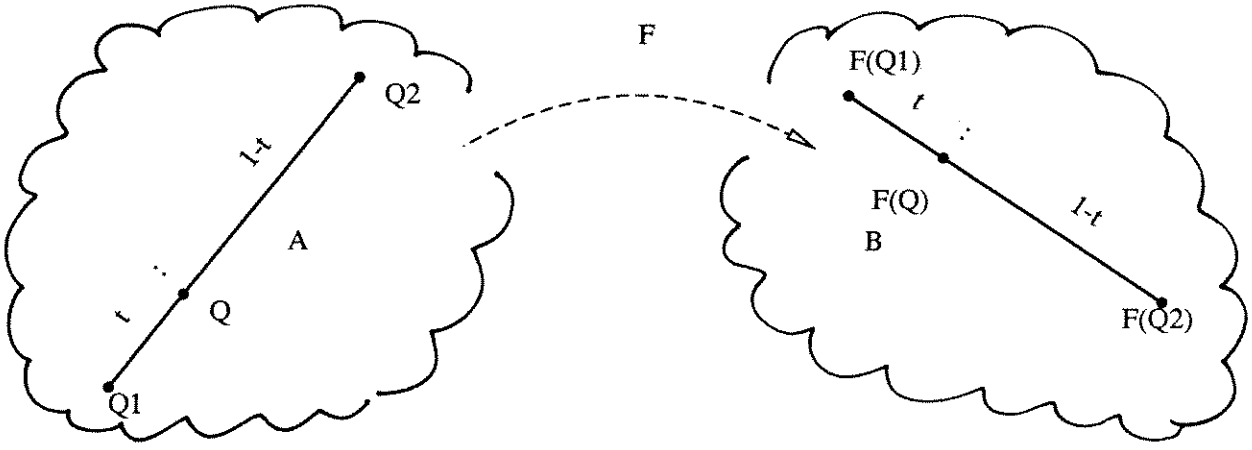

FIGURE 10. The action of an affine map on a line segment.

In the introduction to this section it was claimed that affine transformations carry points to points and vectors to vectors. However, affine transformations are currently defined only on points. Fortunately, we can extend their domains to include the vectors as well. Let $F : \mathcal{A}.\mathcal{P} \mapsto \mathcal{B}.\mathcal{P}$ be an affine map, let $\vec{v}$ be any vector in $\mathcal{A}.\mathcal{V}$, and let P and Q be any

two points in $\mathcal{A}.\mathcal{P}$ such that $\vec{v} = P - Q$. We define $F(\vec{v})$ to be the vector in $\mathcal{B}.\mathcal{V}$ given by $F(P) - F(Q)$. In equation form,

$$F(\vec{v}) \equiv F(P - Q) := F(P) - F(Q).$$

Notice that the points P and Q used in the definition are not unique in that there are many pairs of points whose difference is $\vec{v}$. To verify that the definition of $F(\vec{v})$ is well-formed, it must be shown that if P,Q and P',Q' are two pairs of points whose difference is $\vec{v}$, then $F(P)-F(Q) = F(P')-F(Q')$. This is not difficult to show using the identities of Claim 3.1, so we have left it as an exercise (Exercise 2).

Since the domain of an affine map such as $F : \mathcal{A}.\mathcal{P} \mapsto \mathcal{A}.\mathcal{P}$ can be extended to include $\mathcal{A}.\mathcal{V}$, we consider F being defined on all of $\mathcal{A}$, and hence we write simply $F : \mathcal{A} \mapsto \mathcal{B}$.

Using the definition of the action of an affine map on vectors, it is also not difficult to show that F is a linear transformation on the set of vectors. That is, F satisfies

$$F(u_1\vec{v}_1 + \cdots + u_n\vec{v}_n) = u_1F(\vec{v}_1) + \cdots + u_nF(\vec{v}_n), \tag{3.11}$$

for all $u_1, ..., u_n$ and for all $\vec{v}_1, ..., \vec{v}_n$. The proof of this fact is rather instructive in that it demonstrates a use of the head-to-tail axiom. We will show that F satisfies the following two conditions:

1. $F(\vec{v} + \vec{w}) = F(\vec{v}) + F(\vec{w})$.
2. $F(\alpha\vec{v}) = \alpha F(\vec{v})$.

Equation (3.11) can then be shown by inducting on the number of terms in the sum. To prove condition 1, let P, Q and R be points such that $\vec{v} = Q - R$ and $\vec{w} = P - Q$, as shown in Figure 11. By the head-to-tail axiom, $F(\vec{v} + \vec{w}) = F(P) - F(R)$. Using the head-to-tail axiom again in the range gives the desired result:

$$\begin{aligned} F(\vec{v} + \vec{w}) &= F(P) - F(R) \\ &= [F(P) - F(Q)] + [F(Q) - F(R)] \\ &= F(\vec{v}) + F(\vec{w}). \end{aligned}$$

To prove condition 2, we note that the vector $\alpha\vec{v}$ can be written as

$$\alpha\vec{v} = [(1 - \alpha)R + \alpha Q] - R.$$

The desired result can now be achieved in a just a few steps:

$$\begin{aligned} F(\alpha\vec{v}) &= F([(1 - \alpha)R + \alpha Q] - R) \\ &= F([(1 - \alpha)R + \alpha Q]) - F(R) \\ &= (1 - \alpha)F(R) + \alpha F(Q) - F(R) \\ &= \alpha F(\vec{v}). \end{aligned}$$

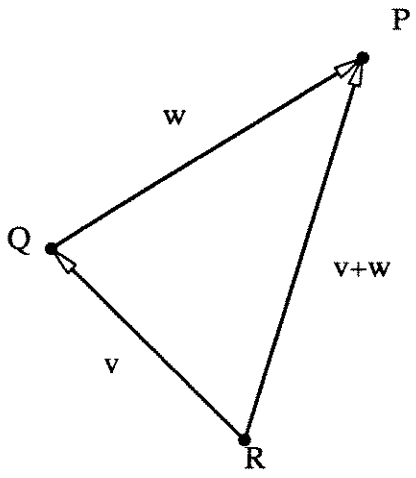

FIGURE 11. The points P,Q, and R in the proof of condition 1.

Next, we show that F also satisfies

$$F(Q + \vec{v}) = F(Q) + F(\vec{v})$$

for every point Q and every vector $\vec{v}$. To do this, let P be such that $\vec{v} = P - Q$. Thus, $F(Q + \vec{v}) = F(Q + P - Q)$. Since the expression $Q + P - Q$ is an affine combination,

$$\begin{aligned} F(Q + P - Q) &= F(Q) + F(P) - F(Q) \\ &= F(Q) + [F(P) - F(Q)] \\ &= F(Q) + F(\vec{v}). \end{aligned}$$

Putting these facts together reveals that F preserves affine coordinates:

$$F(p_1\vec{v}_1 + \cdots + p_n\vec{v}_n + \mathcal{O}) = p_1F(\vec{v}_1) + \cdots + p_nF(\vec{v}_n) + F(\mathcal{O}), \tag{3.12}$$

showing that affine maps are completely determined once the image of a frame is known. This simple observation can be put to practical use. In fact, the geometric ADT uses this fact as the basis for a coordinate-free method of specifying affine transformations.

EXERCISES:

1. Show that the centroids of triangles are mapped to centroids of triangles by affine maps.
2. Show that the definition of the action of F on vectors is well-formed in the sense that if P, Q and P', Q' are two pairs of points such that $P - Q = P' - Q'$, then $F(P) - F(Q) = F(P') - F(Q')$.
3. Let L_1 and L_2 be two lines that pass through the points Q_1, P_1 and Q_2, P_2 respectively. These lines are said to be *parallel* if $P_1 - Q_1$ is a scalar multiple of $P_2 - Q_2$. Show that parallel lines are mapped to parallel lines under affine maps.
4. Show that if $F : \mathcal{A} \mapsto \mathcal{B}$ and $G : \mathcal{B} \mapsto \mathcal{C}$ are affine maps, then the composition $G \circ F$ is also an affine map.

* 3.3.1. Matrix Representations of Affine Transformations

Just as points and vectors can be represented as matrices, so too can affine transformations. For notational simplicity, the following discussion will be restricted to maps between affine planes. This restriction is not limiting since all arguments carry through to affine spaces of arbitrary dimension.

Let $\mathcal{A}$ and $\mathcal{B}$ be two affine planes, let $F : \mathcal{A} \mapsto \mathcal{B}$ be an affine transformation, let $(\vec{v}_1, \vec{v}_2, \mathcal{O}_\mathcal{A})$ be a frame for $\mathcal{A}$, let $(\vec{w}_1, \vec{w}_2, \mathcal{O}_\mathcal{B})$ be a frame for $\mathcal{B}$, and let P be an arbitrary point whose coordinates relative to $(\vec{v}_1, \vec{v}_2, \mathcal{O}_\mathcal{A})$ are $(p_1, p_2, 1)$. We ask: what are the coordinates of $F(P)$ relative to $(\vec{w}_1, \vec{w}_2, \mathcal{O}_\mathcal{B})$? The solution requires little more than simple manipulation. We begin by expanding P in coordinates and use the fact that F preserves affine coordinates:

$$\begin{aligned} F(P) &= F(p_1\vec{v}_1 + p_2\vec{v}_2 + \mathcal{O}_\mathcal{A}) \\ &= p_1F(\vec{v}_1) + p_2F(\vec{v}_2) + F(\mathcal{O}_\mathcal{A}). \end{aligned} \tag{3.13}$$

Since F carries vectors (points) in $\mathcal{A}$ into vectors (points) in $\mathcal{B}$, the quantities $F(\vec{v}_1)$ and $F(\vec{v}_2)$ are vectors in $\mathcal{B}$ and the quantity $F(\mathcal{O}_\mathcal{A})$ is a point in $\mathcal{B}$, and as such they each have affine coordinates relative to the frame $(\vec{w}_1, \vec{w}_2, \mathcal{O}_\mathcal{B})$. Suppose that

$$\begin{aligned} F(\vec{v}_1) &= f_{1,1}\vec{w}_1 + f_{1,2}\vec{w}_2 \\ F(\vec{v}_2) &= f_{2,1}\vec{w}_1 + f_{2,2}\vec{w}_2 \\ F(\mathcal{O}_\mathcal{A}) &= f_{3,1}\vec{w}_1 + f_{3,2}\vec{w}_2 + \mathcal{O}_\mathcal{B} \end{aligned}$$

Using these coordinates, Equation (3.13) can be manipulated as follows:

$$\begin{aligned} F(P) &= \begin{pmatrix} p_1 & p_2 & 1 \end{pmatrix} \begin{pmatrix} F(\vec{v}_1) \\ F(\vec{v}_2) \\ F(\mathcal{O}_\mathcal{A}) \end{pmatrix} \\ &= \begin{pmatrix} p_1 & p_2 & 1 \end{pmatrix} \begin{pmatrix} f_{1,1}\vec{w}_1 + f_{1,2}\vec{w}_2 \\ f_{2,1}\vec{w}_1 + f_{2,2}\vec{w}_2 \\ f_{3,1}\vec{w}_1 + f_{3,2}\vec{w}_2 + \mathcal{O}_\mathcal{B} \end{pmatrix} \\ &= \begin{pmatrix} p_1 & p_2 & 1 \end{pmatrix} \begin{pmatrix} f_{1,1} & f_{1,2} & 0 \\ f_{2,1} & f_{2,2} & 0 \\ f_{3,1} & f_{3,2} & 1 \end{pmatrix} \begin{pmatrix} \vec{w}_1 \\ \vec{w}_2 \\ \mathcal{O}_\mathcal{B} \end{pmatrix} \\ &= \begin{pmatrix} p_1 & p_2 & 1 \end{pmatrix} \mathbf{F} \begin{pmatrix} \vec{w}_1 \\ \vec{w}_2 \\ \mathcal{O}_\mathcal{B} \end{pmatrix} \\ &= \begin{pmatrix} p_1' & p_2' & 1 \end{pmatrix} \begin{pmatrix} \vec{w}_1 \\ \vec{w}_2 \\ \mathcal{O}_\mathcal{B} \end{pmatrix} \end{aligned}$$

Thus, the point P with coordinates $(p_1, p_2, 1)$ gets transformed to the point $F(P)$ with coordinates $(p_1' \ p_2' \ 1)$, where $(\ p_1' \quad p_2' \quad 1\ ) = (\ p_1 \quad p_2 \quad 1\ )\,\mathbf{F}$. For this reason, the matrix $\mathbf{F}$ is called the *matrix representation* of F *relative to the frames* $(\vec{v}_1, \vec{v}_2, \mathcal{O}_A)$ and $(\vec{w}_1, \vec{w}_2, \mathcal{O}_B)$. Notice that

The first row of $\mathbf{F}$ is the representation of $F(\vec{v}_1)$.

The second row of $\mathbf{F}$ is the representation of $F(\vec{v}_2)$.

The third row of $\mathbf{F}$ is the representation of $F(\mathcal{O}_A)$.

As a consequence, affine maps are represented as matrices whose last column is $(0\ 0\ 1)^T$. Conversely, every matrix whose last column is $(0\ 0\ 1)^T$ represents some affine transformation.

EXAMPLE 3.3: As a specific example of the construction of a matrix representation of an affine transformation, consider the construction of a matrix representation of the translation T of Example 3.2. To do this, we must pick frames in both the domain and the range. Since T maps $\mathcal{A}$ onto itself, the domain and range are the same space, so we may as well pick a single frame $(\vec{v}_1, \vec{v}_2, \mathcal{O})$ to serve double-duty. Suppose that in this frame the vector $\vec{t}$ has the coordinates $(a, b, 0)$. All we have to do to determine the matrix $\mathbf{T}$ is to determine what T does to $\vec{v}_1$, $\vec{v}_2$, and $\mathcal{O}$.

Let's do the easy part first. The third row of $\mathbf{T}$ consists of the coordinates of $T(\mathcal{O})$, which are $(a, b, 1)$ since

$$\begin{aligned} T(\mathcal{O}) &= \mathcal{O} + \vec{t} \\ &= a\vec{v}_1 + b\vec{v}_2 + \mathcal{O}. \end{aligned}$$

The first row of $\mathbf{T}$ consists of the coordinates of $T(\vec{v}_1)$. To see what these are, let R and S be two points such that $\vec{v}_1 = R - S$. Then, by definition of how affine maps behave on vectors,

$$\begin{aligned} T(\vec{v}_1) &= T(R - S) && \text{– by def of } R,S \\ &= T(R) - T(S) && \text{– by def } T \text{ on vectors} \\ &= (R + \vec{t}) - (S + \vec{t}) && \text{– by def of } T \\ &= R - S && \text{– by Claim 3.1(f)} \\ &= \vec{v}_1 && \text{– by def of } R,S. \end{aligned}$$

In other words, T does not affect $\vec{v}_1$. In fact, the derivation above works for all vectors, not just $\vec{v}_1$, so T doesn't affect any vector. This means that the first row of $\mathbf{T}$ is $(1\ 0\ 0)$, and the second row is $(0\ 1\ 0)$. Putting this all together, relative to the frame $(\vec{v}_1, \vec{v}_2, \mathcal{O})$, T is represented by the matrix

$$\mathbf{T} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{pmatrix}.$$

REMARK: This is the form for a translation matrix that is typically found in graphics texts. However, in contrast to those texts, the matrix above was arrived at using only affine arguments, not projective ones. □

EXERCISES:

5. Let $F : \mathcal{A} \mapsto \mathcal{B}$ be the unique affine map that carries the frame $\mathcal{F}_\mathcal{A}$ in $\mathcal{A}$ into the frame $\mathcal{F}_\mathcal{B}$ in $\mathcal{B}$. Show that the matrix representation of F relative to $\mathcal{F}_\mathcal{A}$ and $\mathcal{F}_\mathcal{B}$ is the identity matrix. (This implies that the identity matrix does not necessarily represent the identity transformation.)
6. Let $F : \mathcal{A} \mapsto \mathcal{B}$ and $G : \mathcal{B} \mapsto \mathcal{C}$ be affine maps, and let $\mathcal{F}_\mathcal{A}$, $\mathcal{F}_\mathcal{B}$, and $\mathcal{F}_\mathcal{C}$ be frames in $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, respectively. Show that if $\mathbf{F}$ is the matrix representation of F relative to $\mathcal{F}_\mathcal{A}$ and $\mathcal{F}_\mathcal{B}$, and $\mathbf{G}$ is the matrix representation of F relative to $\mathcal{F}_\mathcal{B}$ and $\mathcal{F}_\mathcal{C}$, then $\mathbf{FG}$ is the matrix representation of $G \circ F$.
7. Suppose that an affine map $T : \mathcal{A} \mapsto \mathcal{B}$ has a matrix representation $\mathbf{T}$ relative to frames $\mathcal{F}_\mathcal{A}$ and $\mathcal{F}_\mathcal{B}$, and suppose that $\mathcal{F}'_\mathcal{A}$ is a frame in $\mathcal{A}$ such that coordinates relative to $\mathcal{F}_\mathcal{A}$ are changed into coordinates relative to $\mathcal{F}'_\mathcal{A}$ by multiplying by a matrix $\mathbf{F}$. Show that the matrix representation of T relative to $\mathcal{F}'_\mathcal{A}$ and $\mathcal{F}_\mathcal{B}$ is $F^{-1}\,T$.

3.4. Projective Transformations

In the previous section it was mentioned that perspective projections are not affine transformations. To model perspective, we must generalize to the *projective transformations.*

Affine transformations were shown to carry lines to lines and to preserve ratios of distances along lines. These two properties can in fact be used as the definition of affine transformations. More generally, projective transformations can be defined as maps that carry lines to lines and preserve *cross ratios.* The cross ratio along a line segment $PQRS$ is defined as a ratio of ratios:

$$\text{CrossRatio}(P, Q, R, S) = \frac{PQ : QS}{PR : RS},$$

as depicted in Figure 12.

Since the cross ratio is a generalization of the simple ratio, every affine transformation is also a projective transformation. Although we won't do so here, it is possible to show that the composition of two projective maps yields a projective map; hence, the composition of an affine map with a projective map also yields a projective map.

Even though projective transformations carry points to points,† lines to lines, and more generally, hyperplanes to hyperplanes, they do not preserve the structure of affine spaces

† Actually, some points in the domain can be mapped to points at infinity in the range. These points therefore do not have images in the affine range space.

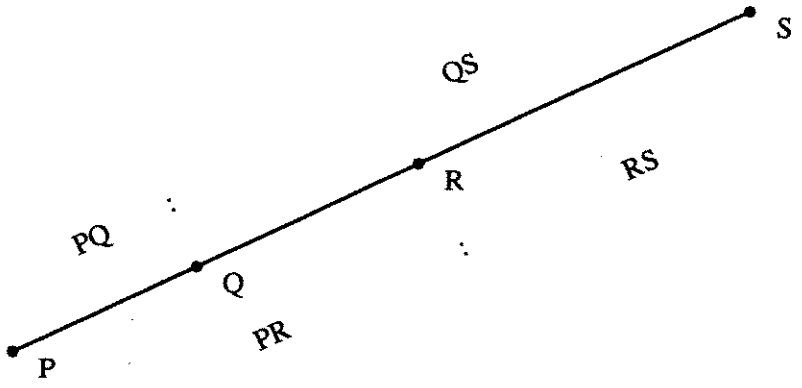

FIGURE 12. Cross ratio.

(they do, however, preserve the structure of projective spaces). In particular, they don't map vectors to vectors. In fact, it is not possible to extend the domain of a projective transformation to include vectors. We might be tempted to offer a definition similar to the one used for affine spaces. That is, suppose T is a projective transformation and suppose P and Q are points such that $\vec{v} = P - Q$. If we define the action of T on $\vec{v}$ by $T(\vec{v}) = T(P - Q) = T(P) - T(Q)$ we run into trouble. Specifically, this definition is not well-formed in that if P', Q' are a pair of points such that $P - Q = P' - Q'$, then in general $T(P) - T(Q) \neq T(P') - T(Q')$, implying that $T(P - Q) \neq T(P' - Q')$, which in turn implies that $T(\vec{v})$ is not well-defined. An explicit example of this difficulty is shown in Figure 13. The point Q is the midpoint of P, R, implying that $P - Q = Q - R$. However, since T does not map Q to the midpoint of $T(P), T(R)$, we find that $T(P) - T(Q) \neq T(Q) - T(R)$.

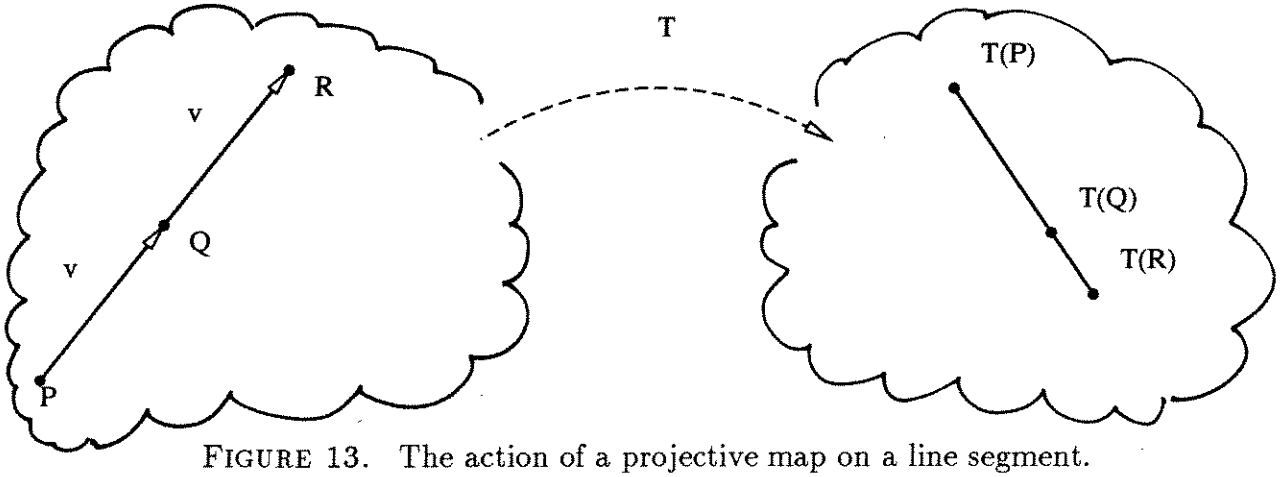

FIGURE 13. The action of a projective map on a line segment.

The fact that projective transformations can't map vectors must be reflected in the geometric algebra and the ADT. The geometric algebra can deal with the situation simply by leaving it undefined; the ADT can handle the problem by signaling a type-clash if a request is made to map a vector through a projective transformation.

In Section 3.3, it was shown that an affine map is completely characterized if its action on a simplex is known. A similar result holds for projective maps. We state here without proof that a projective map from an affine n-space is completely characterized if its action on a n-simplex plus one other point is known [Flohr and Raith '74]. The complete collection of $n+2$ points must be such that if any one of the points is deleted, the remaining $n+1$ points form an n-simplex. A collection of points satisfying this condition are said to be in *general position*. Thus, a projective map defined on an affine plane is completely characterized once its action on a collection of four points in general position is known.

* 3.5. Matrix Representations of Projective Maps

The representation of projective maps using homogeneous matrices has been well documented elsewhere (cf. [Foley and Van Dam '82] [Riesenfeld '81]), but for the sake of completeness, we offer a strenuously abbreviated account for projective maps between affine planes.

Let $T : \mathcal{A} \mapsto \mathcal{B}$ be a projective map between affine planes $\mathcal{A}$ and $\mathcal{B}$, let $\mathcal{F}_\mathcal{A}$ and $\mathcal{F}_\mathcal{B}$ be frames in $\mathcal{A}$ and $\mathcal{B}$. T can be represented by some 3×3 homogeneous matrix $\mathbf{T}$ relative to these frames. A homogeneous matrix has the property that multiplication through by a scalar does not change the represented transformation. Unlike matrix representations of affine transformations, the last column of $\mathbf{T}$ need not be $(\,0 \quad 0 \quad 1\,)^T$.

To see how $\mathbf{T}$ is used in calculation, let P be a point in $\mathcal{A}$ having coordinates $(p_1, p_2, 1)$ relative to $\mathcal{F}_\mathcal{A}$. The coordinates of $P' = T(P)$ relative to $\mathcal{F}_\mathcal{B}$ can be obtained from $\mathbf{T}$ in two steps:

1. Let $(\,x \quad y \quad w\,) = (\,p_1 \quad p_2 \quad 1\,)\,\mathbf{T}$.
2. The coordinates of P' relative to $\mathcal{F}_\mathcal{B}$ are then given by $(\,\frac{x}{w} \quad \frac{y}{w} \quad 1\,)$.

4. Euclidean Geometry

In affine geometry, metric concepts such as absolute length, distance, and angles are not defined. This is demonstrated by the fact that up to this point we have not used these concepts in the development of affine geometry. In fact, the only related concept that has been used is the fact that affine maps preserve relative ratios. However, in graphics and computer aided design, it is often necessary to represent metric information, for without this information it is not possible to define right angles or to distinguish spheres from ellipsoids.

When metric information is added to an affine space, the result is the familiar concept of a Euclidean space. In other words, a Euclidean space is a special case of an affine space

in which it is possible to measure absolute distances, lengths, and angles. Consequently, all results that were obtained for affine spaces and affine maps also hold in Euclidean spaces. For instances, the points of a Euclidean space are closed under affine combinations, parallel lines map to parallel lines under affine maps, and (in a Euclidean plane) every pair of triangles is related by a unique affine map.

4.1. The Inner Product

In keeping with our algebraic approach to geometry, we shall incorporate metric knowledge by introducing a new algebraic entity called an *inner product*. An inner product for an affine space $\mathcal{A}$ is a function that maps a pair of vectors in $\mathcal{A}.\mathcal{V}$ into the reals. Rather than using a notation such as $f(\vec{u}, \vec{v})$ to denote an inner product, we use the more familiar form $< \vec{u}, \vec{v} >$. Such a bi-variate function must possess the following properties to achieve the status of an inner product:

(i) *Symmetry:* For every pair of vectors $\vec{u}, \vec{v}$, $< \vec{u}, \vec{v} >=< \vec{v}, \vec{u} >$.

(ii) *Bi-linearity:* For every $\alpha, \beta \in \Re$ and for every $\vec{u}, \vec{v}, \vec{w} \in \mathcal{A}.\mathcal{V}$,

- $< \alpha\vec{u} + \beta\vec{v}, \vec{w} >= \alpha < \vec{u}, \vec{w} > +\beta < \vec{v}, \vec{w} >$.
- $< \vec{u}, \alpha\vec{v} + \beta\vec{w} >= \alpha < \vec{u}, \vec{v} > +\beta < \vec{u}, \vec{w} >$.

(iii) *Positive Definiteness:* For every $\vec{v} \in \mathcal{A}.\mathcal{V}$, $< \vec{v}, \vec{v} >> 0$ if $\vec{v}$ is not the zero vector, and $< \vec{0}, \vec{0} >= 0$.

A Euclidean space $\mathcal{E}$ can now be defined as an affine space together with a distinguished inner product; that is, $\mathcal{E} = (\mathcal{A}, <, >)$. To comform more closely with standard practice, the inner product associated with a particular Euclidean space will generally be denoted by $\cdot$, and will generally be referred to as the dot product. Thus, we write $u \cdot v$ to stand for $< u, v >$.

The dot product is used to define length, distance, and angles as follows:

- *The length of a vector.*

$$|\vec{v}| := \sqrt{\vec{v} \cdot \vec{v}}.$$

- *The distance between two points.*

$$\text{Dist}(P, Q) := |P - Q| .$$

- *The angle between two vectors.*

$$\text{Angle}(\vec{v}, \vec{w}) := \cos^{-1}\left(\frac{\vec{v} \cdot \vec{w}}{|\vec{v}|\,|\vec{w}|}\right).$$

Associated with every vector $\vec{v}$ is a unique vector $\hat{v}$ having unit length that points in the same direction as $\vec{v}$. The vectors $\vec{v}$ and $\hat{v}$ are, of course, related by

$$\hat{v} := \frac{\vec{v}}{|\vec{v}|}.$$

The definition of angles allows us to define notion of *perpendicularity* or *orthogonality.* In particular, two vectors $\vec{v}$ and $\vec{w}$ are said to be perpendicular (or orthogonal) if $\vec{v} \cdot \vec{w} = 0$. We can also define the vectors to be *parallel* if $\hat{v} \cdot \hat{w} = 1$, and *anti-parallel* if $\hat{v} \cdot \hat{w} = -1$.

The notions of unit vectors and orthogonality allow us to identify an important kind of coordinate frame for Euclidean spaces. A coordinate frame $(\vec{e}_1, ...\vec{e}_n, \mathcal{O})$ is said to be a *Cartesian frame* if the basis vectors are *ortho-normal*; that is, if the basis vectors satisfy

$$\vec{e}_i \cdot \vec{e}_j = \begin{cases} 1 & \text{if } i = j; \\ 0 & \text{otherwise.} \end{cases}$$

In the important special case of Euclidean 3-spaces, it is convenient to define another operation on vectors, namely the *cross product.* Given a pair of vectors $\vec{v}$ and $\vec{w}$ from a Euclidean 3-space, we define $\times$ by the equation

$$\vec{v} \times \vec{w} = |\vec{v}|\,|\vec{w}| \sin\theta \hat{n},$$

where θ is the angle between the vectors and $\hat{n}$ is the unique unit vector that is perpendicular to $\vec{v}$ and $\vec{w}$ such that $\vec{v}$, $\vec{w}$ and $\hat{n}$ satisfy the "right hand rule."

4.2. Normal Vectors and the Dual Space

In many graphics and modeling applications it is convenient to introduce the idea of a *normal vector.* For instance, in polygonal modeling it is common to represent objects by polyhedra where each vertex is tagged with a normal vector that is used in Gouraud or Phong shading. Normal vectors are also important for ray tracing applications since the surface normal determines the direction of a reflected ray, and is one of the determining factors in the direction of a refracted ray.

Unfortunately, the term "normal vector" implies that these objects behave just like other vectors. While this is nearly correct, there are important situations where subtleties can occur. A simplified situation is shown in Figure 14 for a hypothetical two-dimensional polygonal modeling application. The left portion of Figure 14 represents the definition space of a polygonal approximation to a circle. The right portion of the figure is the image of the polygon under the indicated non-uniform scaling. Notice that if the normal vectors are transformed as vectors, then their images don't end up being perpendicular to the image of the circle (an ellipse). In fact, they have become more horizontal when they should have become more vertical. This rather unexpected behavior can also occur in ray tracers that model objects hierarchically by applying transformations to subobjects when instancing them into higher level objects. The visual effect is to produce incorrect shading and reflections.

Fortunately, there is a remedy that is firmly rooted in the fundamentals of geometry. The idea is to represent normals not as vectors, but as algebraic entities called *dual vectors,* the definition of which we shall present shortly. Intuitively, dual vectors represent oriented

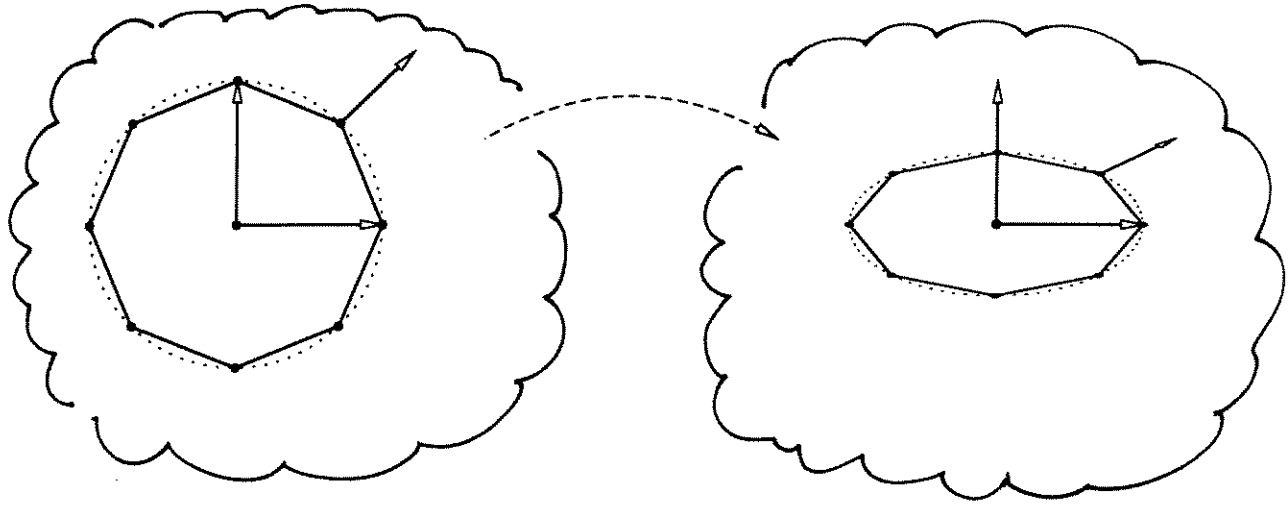

FIGURE 14. Normals transforming as vectors.

planes. By modeling the oriented plane directly, it will be possible to construct a transformation rule that guarantees that normals map in a way that guarantees perpendicularity with the appropriate plane. To make these idea precise, we must take a short excursion into the concepts of linear functionals and dual spaces.

WARNING: The discussion to follow is intended for the purist who is interested in the algebraic details of dual spaces and how they relate to normals; for those interested primarily in results, the remainder of this section should probably be skipped, at least on first reading. The results of this section can be summarized as follows:

— Dual vectors are represented as column matrices.

— If $\mathbf{F}$ is a matrix representation of an affine map (relative to a pair of Cartesian frames), then dual vectors are transformed by applying the inverse of $\mathbf{F}$ on the left, followed by setting the last component to zero.

For the moment, let us leave the realm of affine and Euclidean geometry and work instead in the context of vector spaces. A *linear functional* λ on a vector space $\mathcal{V}$ is a map from $\mathcal{V}$ into the reals that satisfies the linearity condition

$$\lambda(\alpha\vec{v} + \beta\vec{w}) = \alpha\lambda(\vec{v}) + \beta\lambda(\vec{w}),$$

for all $\vec{v}, \vec{w} \in \mathcal{V}$ and for all $\alpha, \beta \in \Re$. It turns out that the set of all linear functionals on a vector space $\mathcal{V}$ itself forms a vector space, generally denoted by $\mathcal{V}^*$ (cf. [O'Nan '76]). The vector space $\mathcal{V}^*$ of linear functionals is called the *dual space* of $\mathcal{V}$. To reinforce the dual nature of the spaces $\mathcal{V}$ and $\mathcal{V}^*$, the elements of $\mathcal{V}$ are more accurately known as *primal vectors* and the elements of $\mathcal{V}^*$ are called *dual vectors.*

An inner product on the vector space can be used to establish an association between primal vectors and dual vectors. In particular, using the bracket notation for the inner

product, if $\vec{v}$ is held fixed, the expression $< \vec{v}, \vec{u} >$ is a linear functional whose argument is $\vec{u}$; that is, $\lambda(\vec{u}) :=< \vec{v}, \vec{u} >$ is a linear functional on $\mathcal{V}$ and is therefore a dual vector (associated with the vector $\vec{v}$). To avoid having to invent a symbol to act as the argument $\vec{u}$, it is more common to write $\lambda :=< \vec{v}, >$. Using this association, we can define the functional $< \vec{v}, >$ to be the dual of $\vec{v}$. In equation form,

$$\mathcal{D}_{\vec{v}} :=< \vec{v}, > .$$

In this form we recognize that $\mathcal{D}$ is actually a linear mapping from $\mathcal{V}$ to $\mathcal{V}^*$. In fact, $\mathcal{D}$ is one-to-one and onto, implying that it is also invertible. The definition of $\mathcal{D}$ provides another interpretation of the quantity $< \vec{v}, \vec{w} >$. By construction, $< \vec{v}, \vec{w} >= \mathcal{D}_{\vec{v}}(\vec{w})$, implying that $< \vec{v}, \vec{w} >$ can be obtained by first dualizing $\vec{v}$, then applying the resulting linear functional to $\vec{w}$.

It was mentioned in the introduction to this section that dual vectors represent oriented planes. To be more precise, dual vectors represent oriented hyperplanes. To see this, notice that $\mathcal{D}_{\vec{v}}(\vec{w})$ vanishes whenever $\vec{w}$ is perpendicular to $\vec{v}$ since perpendicularity implies that $< \vec{v}, \vec{w} >= 0$. Recall that in a vector space the set of vectors perpendicular to a fixed vector forms a hyperplane that contains the zero vector. Thus, the linear functional $\mathcal{D}_{\vec{v}}$ represents the hyperplane through the origin perpendicular to $\vec{v}$. This hyperplane is oriented because we can distinguish a positive and negative side. The vector $\vec{w}$ is on the positive side of the hyperplane if $\mathcal{D}_{\vec{v}}(\vec{w}) > 0$; it is on the negative side if $\mathcal{D}_{\vec{v}}(\vec{w}) < 0$.

To translate the above results from vectors in a vector space into a Euclidean setting, we observe that the freedom of vectors to move about in Euclidean space means that a dual vector defines only the orientation of the hyperplane, but does not fix it absolutely in space. The hyperplane can be fixed by specifying a point through which the hyperplane must pass. Thus, in a Euclidean space an oriented hyperplane is represented as a point together with a dual vector. For instance, the hyperplane defined by a point P and a dual vector $\mathcal{D}_{\vec{v}}$ is the set of points $\{Q|\mathcal{D}_{\vec{v}}(Q-P)\} = 0$. The point Q is on the positive (negative) side of the hyperplane if $\mathcal{D}_{\vec{v}}(Q-P)$ is positive (negative), as shown in Figure 15 for a Euclidean 3-space.

In preparation for Section 5.3.2, we observe that hyperplanes represented by a point and a vector can also be thought of as *affine functionals* defined on points (an affine functional is an affine map from an affine space into the reals). To see this, consider the oriented hyperplane passing through a point P normal to a vector $\vec{v}$. We define a linear functional π by:

$$\pi(Q) := \mathcal{D}_{\vec{v}}(Q-P), \qquad Q \in \mathcal{A}.\mathcal{P}.$$

Geometrically, the value $\pi(Q)$ measures the (signed) perpendicular distance from the point Q to the hyperplane. Specifically, if d_Q represents the true signed perpendicular distance

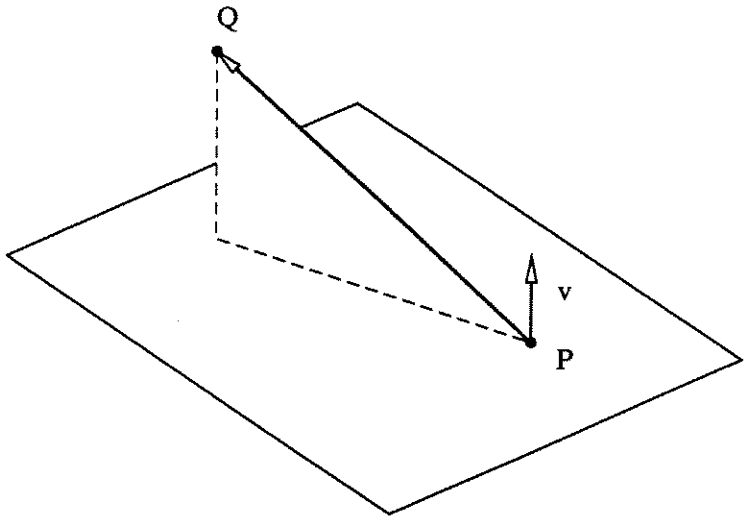

FIGURE 15. The representation of a plane as a vector and a point.

from Q to the hyperplane, then

$$\begin{aligned}\pi(Q) &= \mathcal{D}_{\vec{v}}(Q-P)\\ &= \vec{v}\cdot(Q-P)\\ &= |\vec{v}|\,|Q-P|\cos\theta\\ &= |\vec{v}|\,d_Q,\end{aligned}$$

showing that $\pi(Q)$ is a fixed scalar multiple of the true distance. Notice that if $\vec{v}$ is a unit vector, then $\pi(Q)$ is precisely the perpendicular distance to the plane.

At this point it is not clear that we have gained any new insight from the introduction of dual spaces and dual vectors. After all, one could interpret the above discussion as saying nothing more than a plane is defined by a point P and a vector $\vec{v}$. The real power of the dual vector approach is in the determination of how dual vectors, and hence planes and hyperplanes, transform under affine maps.

Let $F : \mathcal{A} \mapsto \mathcal{B}$ be an invertible affine map and let $\vec{v}$ be a vector in $\mathcal{A}.\mathcal{V}$. We would like to extend the domain of F to include the dual vectors such a way that perpendicularity is preserved. This goal can be achieved if we define the action of F on a dual vector $\mathcal{D}_{\vec{v}}$ as

$$F(\mathcal{D}_{\vec{v}}) := \mathcal{D}_{\vec{v}} \circ F^{-1}.$$

To see that perpendicularity is preserved with this definition, let $\vec{w}$ be any non-zero vector in $\mathcal{A}.\mathcal{V}$, let $\vec{w}'$ be its image under F; similarly, let $\mathcal{D}'_{\vec{v}}$ be the image of $\mathcal{D}_{\vec{v}}$ under F. A

consequence of the definition is that $\mathcal{D}_{\vec{v}}(\vec{w}) = \mathcal{D}'_{\vec{v}}(\vec{w}')$, since

$$\begin{aligned}\mathcal{D}_{\vec{v}}(\vec{w}) &=< \vec{v}, \vec{w} > \\ &=< \vec{v}, F^{-1} \circ F\vec{w} > \\ &=< \vec{v}, F^{-1}\vec{w}' > \\ &= \mathcal{D}'_{\vec{v}}(\vec{w}').\end{aligned}$$

Thus, if $\vec{w}$ lies in the hyperplane defined by $\mathcal{D}_{\vec{v}}$ (ie, $\mathcal{D}_{\vec{v}}(\vec{w}) = 0$), then $\vec{w}'$ will lie in the hyperplane defined by $\mathcal{D}'_{\vec{v}}$ (ie, $\mathcal{D}'_{\vec{v}}(\vec{w}') = 0$).

REMARK: As another remark for the purist, we note that dual vectors, i.e., linear functionals, are an instance of the notion of a *covariant tensor*. More specifically, dual vectors are covariant tensors of order one. In general, a covariant tensor of order k is a k-linear map from a vector space into the reals (cf. [Spivak '79]); that is, a map $T(\vec{v}_1, ..., \vec{v}_k)$ is a covariant tensor of order k if it is linear in each of its arguments. Tensors are useful objects in fields such as differential geometry, continuum mechanics, and relativity theory. Ron Goldman has observed that programs designed to solve problems in these areas might therefore benefit from having tensors included in the algebra and the ADT. □

* 4.3. Matrix Representations of Dual Vectors

Rather than representing dual vectors as row matrices as was done for points and vectors, dual vectors are most naturally represented as column matrices. This is most easily seen by considering the coordinate computation of the quantity $\mathcal{D}_{\vec{v}}(\vec{w}) =< \vec{v}, \vec{w} >$. If we expand $\vec{v}$ and $\vec{w}$ into their coordinates relative to a Cartesian frame $(\vec{e}_1, ..., \vec{e}_n, \mathcal{O})$, we find that

$$\mathcal{D}_{\vec{v}}(\vec{w}) =< v_1\vec{e}_1 + \cdots + v_n\vec{e}_n,\ w_1\vec{e}_1 + \cdots + w_n\vec{e}_n > .$$

Bi-linearity is used to rewrite this as

$$\begin{aligned}\mathcal{D}_{\vec{v}}(\vec{w}) = v_1w_1 < \vec{e}_1, \vec{e}_1 > + v_2w_2 < \vec{e}_2, \vec{e}_2 > + \cdots + v_nw_n < \vec{e}_n, \vec{e}_n > \\ + \text{cross terms of the form } v_iw_j < \vec{e}_i, \vec{e}_j > \text{ where } i \neq j.\end{aligned} \tag{4.1}$$

Since the basis vectors are ortho-normal, all cross terms vanish leaving

$$\mathcal{D}_{\vec{v}}(\vec{w}) = v_1w_1 + \cdots + v_nw_n.$$

Notice that this computation can be written in matrix form as the product of a row vector and a column vector: $\mathcal{D}_{\vec{v}}(\vec{w}) = (\, w_1 \quad \cdots \quad w_n \quad 0\,)(\, v_1 \quad \cdots \quad v_n \quad 0\,)^T$. The row vector is the matrix representation of $\vec{w}$ relative to $(\vec{e}_1, ..., \vec{e}_n, \mathcal{O})$, so we can interpret the column vector as the matrix representation of $\mathcal{D}_{\vec{v}}$ relative to the same frame. The multiplication of these matrices corresponds to the application of $\mathcal{D}_{\vec{v}}$ on $\vec{w}$, and hence results in the value $\mathcal{D}_{\vec{v}}(\vec{w})$.

Recall that $\mathcal{D}$ was defined as a linear mapping between a vector space and its dual space. The fact that a vector $\vec{v}$ having coordinates $(v_1, ..., v_n, 0)$ is represented by the row matrix $(\, v_1 \;\; \cdots \;\; v_n \;\; 0\,)$ has a dual $\mathcal{D}_{\vec{v}}$ represented by the column matrix $(\, v_1 \;\; \cdots \;\; v_n \;\; 0\,)^T$ implies that the mapping $\mathcal{D}$ is realized by the matrix transpose operator. Since the transpose operator is its own inverse, the inverse of $\mathcal{D}$ is also realized by the matrix transpose operator.

Given that dual vectors can be represented as column matrices, we now consider the question of how these matrices transform under the action of affine maps. More precisely, let $F : \mathcal{A} \mapsto \mathcal{B}$ be an affine map whose matrix representation relative to Cartesian frames in $\mathcal{A}$ and $\mathcal{B}$ is $\mathbf{F}$, and let $\mathbf{w}$ and $\mathbf{v}$ be the row and column matrices, respectively, that represent $\vec{w}$ and $\mathcal{D}_{\vec{v}}$ relative to the chosen basis. Similarly, let $\mathbf{w}'$ and $\mathbf{v}'$ be the matrix representations of the images of $\vec{w}$ and $\mathcal{D}_{\vec{v}}$ under $\mathbf{F}$. With these definitions, it is $\mathbf{v}'$ that we seek. This column vector can be otained by exanding $\mathcal{D}_{\vec{v}}(\vec{w})$ in matrix notation:

$$\begin{aligned} \mathcal{D}_{\vec{v}}(\vec{w}) &= \mathbf{w}\,\mathbf{v} \\ &= \mathbf{w}\,\mathbf{F}\,\mathbf{F}^{-1}\,\mathbf{v} \\ &= \mathbf{w}'\,\mathbf{F}^{-1}\,\mathbf{v} \end{aligned} \tag{4.2}$$

It was shown earlier that $\mathcal{D}_{\vec{v}}(\vec{w})$ was invariant under affine maps, implying that $\mathcal{D}_{\vec{v}}(\vec{w}) = \mathcal{D}'_{\vec{v}}(\vec{w}') = \mathbf{w}'\,\mathbf{v}'$. Comparing this with the last line of Equation (4.2) reveals that

$$\mathbf{w}'\,\mathbf{F}^{-1}\,\mathbf{v} = \mathbf{w}'\,\mathbf{v}',$$

which can be rewritten as

$$\mathbf{w}'\,\left(\mathbf{F}^{-1}\,\mathbf{v} - \mathbf{v}'\right) = 0.$$

If $\mathbf{w}'$ were a totally arbitrary row vector we could use non-singularity of $\mathbf{F}$ to deduce that

$$\mathbf{v}' = \mathbf{F}^{-1}\,\mathbf{v}. \tag{4.3}$$

There is, however, a problem with taking Equation (4.3) to be the matrix expression for the transformation of dual vectors.† Consider the case when F is a translation, meaning that $\mathbf{F}$ is of the form given in Example 3.3. The inverse matrix is therefore

$$\mathbf{F}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -a & -b & 1 \end{pmatrix}.$$

Using this matrix in Equation (4.3) can result in $\mathbf{v}'$ having a non-zero last component, meaning that it cannot be used as the representation of a dual vector. The solution to this dilemma is hidden in the fact that $\mathbf{w}'$ is not totally arbitrary because its last component must be zero. This means that all but the last component of $\mathbf{v}'$ must agree with $\mathbf{F}^{-1}\,\mathbf{v}$, but

† Thanks to Richard Bartels for pointing this out.

the last component can, without loss of generality, be chosen to be zero. This choice can be forced by introducing a $n+1 \times n+1$ matrix $\mathbf{Z}$ that contains all zero elements except for ones on the first n diagonals. We therefore take as our transformation rule for dual vectors the matrix expression

$$\mathbf{v}' = \mathbf{Z}\,\mathbf{F}^{-1}\,\mathbf{v}. \tag{4.4}$$

One further caveat is in order: Equation (4.4) is only valid when coordinates are expressed relative to Cartesian frames. If coordinates are expressed in a non-Cartesian frame, the transformation rule becomes slightly more complicated. The simplification occurs for Cartesian frames because the cross-terms in Equation (4.1) are guaranteed to vanish. We leave the generalization to arbitrary frames as an exercise for the reader.

EXERCISES:

8. Derive the matrix transformation rule for dual vectors when arbitrary frames in $\mathcal{A}$ and $\mathcal{B}$ are chosen.

5. A Geometric Abstract Data Type

An abstract data type is a collection of data types together with a collection of operations or procedures that manipulate instances of the types. The abstract data type presented here for geometric programming is based on the geometric algebra described in previous sections. In particular, the objects in the algebra are the types of the abstract data type, and the operations in the algebra are the procedures of the abstract data type.

The supported data types are: `Scalar`, `Space`, `Point`, `Vector`, `Normal`, `Frame`, `AffineMap`, and `ProjectiveMap`. Except for the `Space` data type, the connection between these types and the geometric objects in the algebra should be apparent. The `Space` datatype corresponds to a Euclidean space (as opposed to the more general affine space) of the algebra. A relatively complete listing of the procedures of the abstract data type are given in Appendix 2.

Use of the geometric ADT necessarily changes the style in which geometric programs are written. This occurs because applications programmers are generally forbidden from performing coordinate calculations. They must instead use the geometric operations provided by the ADT. Perhaps the best way to demonstrate the use of the ADT is by considering the construction of several sample applications: a simple wire frame display program and a portion of a ray tracer. These applications were chosen because they exhibit many of the style differences that result from use of the ADT.

Before describing the implementation of these applications in Sections 5.3 and 5.4, we first describe the general philosophy of the ADT in Section 5.1; in Section 5.2, we then show how the ambiguity problem is solved using the ADT.

5.1. General Workings

The discussion of the ADT assumes a working knowledge of the C programming language. C was chosen for the following discourse because it seems to be the most wide-spread implementation language in graphics and CAGD. Unfortunately, C is not the most ideal language for geometric programming since it is not possible to overload arithmetic operators, as is desirable when manipulating points, vectors, and so on. More appropriate are languages such as C++ [Stroustrup '86] that do allow operator overloading.

The ADT provides for the creation of Euclidean spaces using the `SCreate` procedure. The calling sequence in our C implementation is:

```
Space SCreate( name, dim)
char *name;
int dim;
```

where `name` is a character string used for debugging purposes, and `dim` is the dimension of the space to be created. When a Euclidean space `S` is created, it comes pre-equipped with a Cartesian frame referred to as `StdFrame(S)` (in the case of a three dimensional space, the frame is defined to be right handed). This frame is used to boot-strap the creation of points, vectors, other frames, and the like. For example, the code fragment shown in Figure 16 creates a Euclidean 3-space called `World`, then constructs a new Cartesian frame called `Camera`. The `Camera` frame is constructed so that its origin is at the point `Eye`, and its basis vectors form a "left handed" Cartesian frame as shown in Figure 17.

```
#define WorldFrame StdFrame(World)

Space World;
Point Eye;
Vector View, ViewX, ViewY, ViewZ;
Frame Camera;

World = SCreate( "World", THREESPACE);

/* Build the viewing frame */
Eye    = PCreate( WorldFrame, 1, 2.5, 2);
View   = VCreate( WorldFrame, -1, -2.5, -2);
ViewZ  = VNormalize( View);
ViewX  = VNormalize(VVCross( ViewZ, Fz( WorldFrame)));
ViewY  = VVCross(ViewX, ViewZ);
Camera = FCreate("Camera", Eye, ViewX, ViewY, ViewZ);
```

FIGURE 16. The construction of a viewing (camera) frame.

Notice that the point `Eye` and the vector `View` are defined by giving their coordinates relative to the standard frame in the `World` space. Even though they were created by using coordinates, once created, they take on a meaning that is independent of coordinate systems. The programmer is therefore freed from having to remember which coordinate system they are represented in. Moreover, since all geometric entities are tagged with the space in which they "live", the system can do a substantial amount of type checking. For instance, before `FCreate` returns the new frame, it checks to see that the origin and the vectors live in the same space and that the vectors form a basis.

To demonstrate that points and vectors have meaning that transcend coordinate systems, let's create two points `Q1` and `Q2` in the `World` space by giving their coordinates relative to

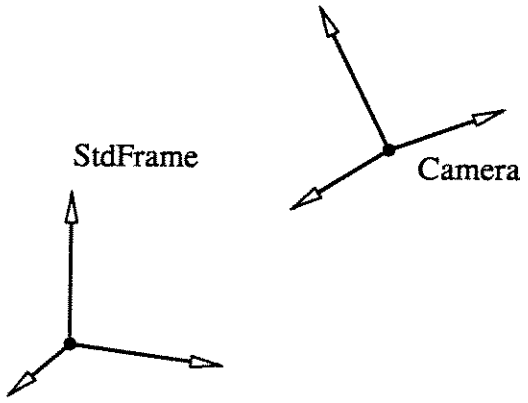

FIGURE 17. Relationship between Camera frame and the standard frame.

different frames:

```
Q1 = PCreate( StdFrame(World), 10, 1, 0);
Q2 = PCreate( Camera, 0, 0, 1);
```

These points can now be treated without regard frames. For instance, if we ask for the distance between these points:

```
d = PPDist(Q1, Q2);
```

we obtain the geometrically correct value of 2.256.

The use of `PCreate` above shows that points can be created by giving their coordinates relative to arbitrary frames. Conversely, coordinates of points, vectors, and so forth can also be extracted relative to arbitrary frames. For instance, the coordinates of `Q1` relative to the `Camera` frame can be extracted and placed in the variables `Qx`, `Qy`, and `Qz` as follows:

```
PCoords( Q1, Camera, &Qx, &Qy, &Qz);
```

Affine and projective maps also have coordinate-independent meanings. For instance, suppose it is desired to create a two dimensional transformation that represents rotation about an arbitrary point P through an angle *theta*. This is easily accomplished using the procedure `ACreate`. First, create a new frame called `RotateFrame` having its origin at P, and x and y vectors inherited from the standard frame in the containing space (see Figure 18).

To use `ACreate` we must determine the image of the elements of `RotateFrame` under the rotation. The following C procedure accomplishes the desired task:

```
/*
** Return the transformation representing rotation about
** point P by an angle theta.
*/
AffineMap Rotate2D( P, theta)
Point P;
Scalar theta;
{
    Frame RotateFrame;
    Vector stdx, stdy, xprime, yprime;

    /* Build the rotation frame */
    stdx = Fx( StdFrame( InSpace(P)));
    stdy = Fy( StdFrame( InSpace(P)));
    RotateFrame = FCreate( "Rotate", P, stdx, stdy);

    /* Build the images of stdx and stdy under the transformation */
    xprime = VCreate( RotateFrame, cos(theta), sin(theta));
    yprime = VCreate( RotateFrame, -sin(theta), cos(theta));

    /* Build and return the transformation */
    return ACreate( RotateFrame, P, xprime, yprime);
}
```

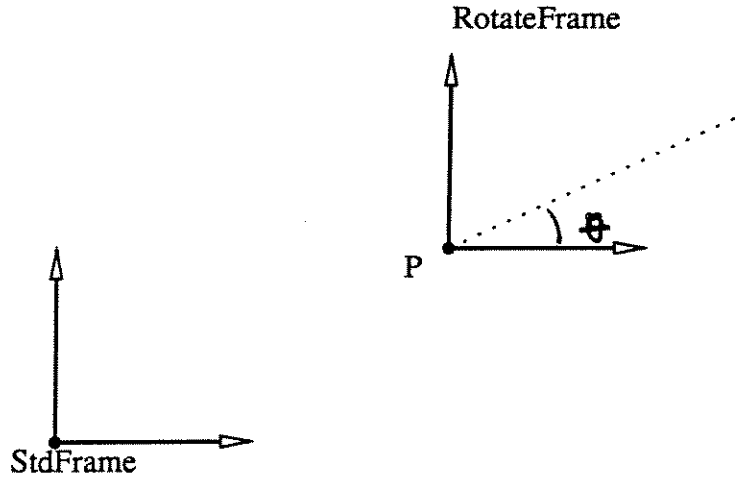

FIGURE 18. Rotation about a point P.

5.2. Ambiguity Revisited

In Section 1 it was claimed that the ADT solves the ambiguity problem in that a given code fragment can have one and only one geometric interpretation. As a demonstration of how this is accomplished, we refer again to the code fragment of Figure 1, the geometric interpretations of which are shown in Figure 2.

Using the ADT, each geometric interpretation is unambiguously reflected in the code. For instance, if the programmer intended a change of coordinates as indicated by Figure 2(a), the appropriate code fragment would be something like:

```
Frame frame1, frame2;
Point P;
Scalar px, py;
   ...
   P = PCreate( frame1, p1, p2);
   PCoords( frame2, &px, &py);
```

where `frame1` and `frame2` are two frames having the geometric relationship indicated in the figure. If the programmer was instead intending to effect a transformation on the space as indicated by Figure 2(b), the appropriate code would be something like:

```
AffineMap T;
Space S;
Point P;
   ...
   T = AScale( FOrg(StdFrame(S)), 2, 1);
   ...
   P = PAxform( P, T);
   ...
```

Finally, if a transformation between separate spaces is to be applied as indicated by Figure 2(c), the code would be something like:

```
AffineMap T;
Space S1, S2;
Point P, Oprime;
Vector xprime, yprime;
   ...
   Compute Oprime, xprime, and yprime in S2
   ...
   T = ACreate( StdFrame(S1), Oprime, xprime, yprime);
   ...
   P = PAxform( P, T);
   ...
```

To reiterate, each of the code fragments above has an unambiguous geometric interpretation that is undeniably apparent from the code. The fact that identical matrix computations are being performed at a lower level is invisible (and irrelevant).

5.3. Example 1: A Simple Wire Frame Display Program

In this section, we consider the construction of a program to display three-dimensional line segments in orthographic projection using the geometric ADT. The program receives its input from a file that contains the world coordinates of the line segment endpoints. Also input to the program are several viewing parameters (see Figure 19):

- A `Camera` frame specification given as an eye point and a viewing direction vector.
- The width and height of the *window* on the projection plane. These parameters are assumed to be stored in variables `wwdith` and `wheight`.
- A viewport specification given as the width, height, and coordinates of the center of the viewport (all given in *normalized device coordinates* [Foley and Van Dam '82]). These parameters are assumed to be stored in variables called `vpwidth`, `vpheight`, `vpx`, and `vpy`.

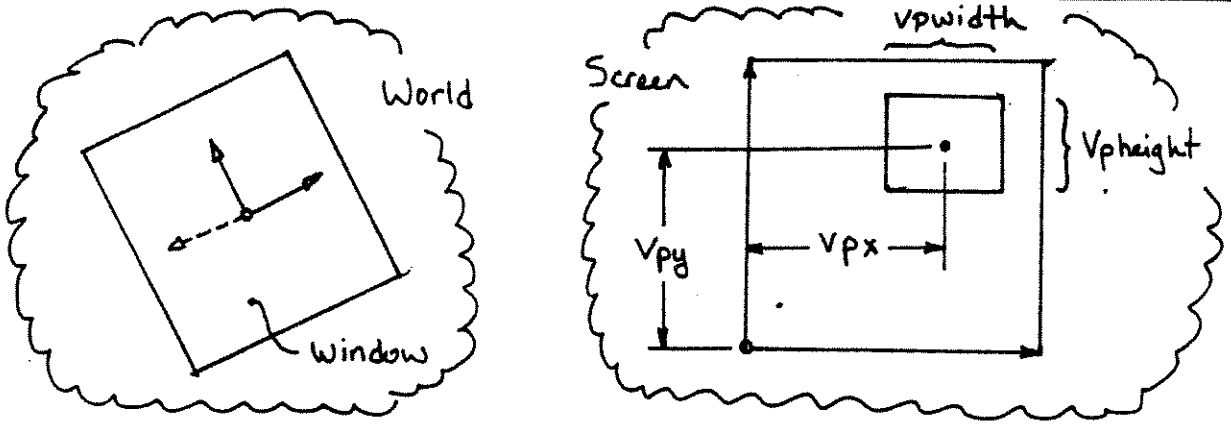

FIGURE 19.. Viewing and viewport specifications.

The general strategy is to create two spaces, `World` and `Screen`. The `World` space is a Euclidean three space in which the objects to be viewed are placed. The `Screen` space corresponds to the physical frame buffer, with the visible portion of the Screen space defined to be the unit square subtended by the standard frame in `Screen` space (see Figure 19). A viewing frame (called `Camera`) is constructed from the viewing parameters as shown earlier in Figure 16, and a clipping volume is constructed about the viewing frame.

Line segments are then processed in four steps:

1. *Point Creation:* When a pair of endpoint coordinates are read from the data file, they are immediately converted to `Points` in the `World` space.

2. *Clipping:* The line segment between the newly created endpoints is clipped to the clipping volume.
3. *Transformation to Screen Space:* The clipped line is (affinely) mapped into the `Screen` space.
4. *Scan Conversion:* The device coordinates of the clipped, projected line segment are extracted and scan converted using a line drawing algorithm such as Bresenham's algorithm [Foley and Van Dam '82].

We shall now examine each of these steps in more detail to demonstrate their implementation using the geometric ADT.

5.3.1. Point Creation

The creation of points given coordinates has already been discussed. However, for completeness, step 1 can be implemented by a C procedure `ReadSegment`:

```
typedef struct {
    Point p1, p2;
} Segment;

/*
** Read the world coordinates of two points, and return a
** Segment structure. No check is done for end-of-file.
*/
Segment ReadSegment()
{
    Scalar x1, y1, z1, x2, y2, z2;
    Segment seg;

    scanf("%lf %lf %lf %lf %lf %lf",
                &x1, &y1, &z1, &x2, &y2, &z2);
    seg.p1  = PCreate( WorldFrame, x1, y1, z1);
    seg.p2 = PCreate( WorldFrame, x2, y2, z2);
    return seg;
}
```

5.3.2. Clipping

Before addressing clipping directly, we first describe how the clipping volume is represented. The clipping volume is represented as a collection of oriented planes. Each plane is represented as a point and a normal (or as described in Section 4.2, as affine functionals), with the orientation of the normal chosen so that the interior of the clipping volume corresponds to the negative side of the plane (the negative half-space). In our implementation, planes are considered as hyperplanes and are represented using the C structure

```
typedef struct {
    Point b;
    Normal n;
} Hyperplane;
```

The clipping volume is then defined to be the intersection of the negative half-spaces defined by the clipping boundary planes. Lines can be clipped to volumes defined in this way by successively clipping against each of the planes. It is therefore sufficient to describe the clipping of a line segment against a single plane.

The clipping of line segments to planes making explicit use of coordinates is well described in standard graphics texts (cf. [Foley and Van Dam '82]). Our approach will, of course, be coordinate-free. First, we shall find it desirable to think of hyperplanes as affine functionals (see Section 4.2), so it is convenient to define a procedure that evaluates the affine functional given a point. In C,

```
/*
** Evaluate the affine functional associated with the Hyperplane Pi
** at the point q.
*/
Scalar EvalHyperplane(Pi, q)
Hyperplane Pi;
Point q;
{
    return NVApply( Pi.n, PPDiff(q,Pi.b));
}
```

Descriptions of the routines `NVApply` and `PPDiff` can be found in Appendix 2.

The coordinate-free clipping of a line segment to a plane can now proceed by evaluating the affine functional at each of the endpoints; call these numbers d_1 and d_2, respectively. Recall from Section 4.2 that d_1 and d_2 are scalar multiples of the signed distance of the endpoints to the plane. If both d_1 and d_2 are positive, the segment is entirely outside the

clipping volume. This corresponds to the traditional *trivial reject* condition of the Cohen-Sutherland algorithm. If both d_1 and d_2 are negative or zero, the segment is entirely inside the clipping volume. This corresponds the traditional *trivial accept* condition. The final case occurs if d_1 and d_2 have differing signs, implying that the segment intersects the plane at some point I. Referring to Figure 20, the point I breaks the line segment into relative ratios $|d_1| : |d_2|$, as is easy to show using similar triangles.

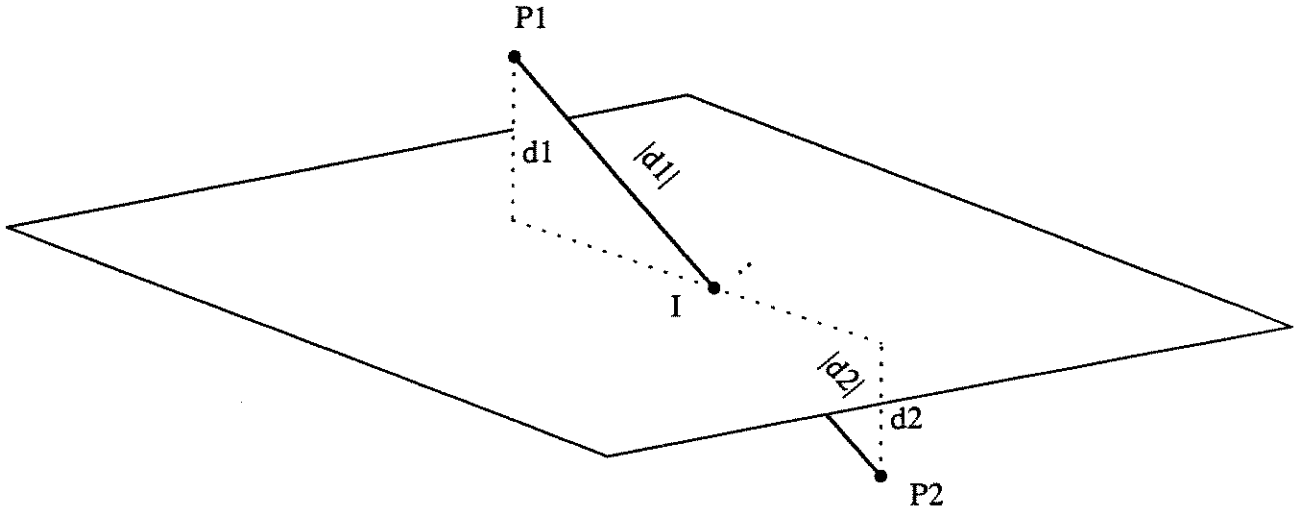

FIGURE 20. Computing the intersection point I.

A C implementation of this algorithm is:

```
/*
** Clip the line segment seg against a Hyperplane Pi, placing the
** result in clipped.
**
** Return TRUE if clipped is not the null segment; return FALSE otherwise.
*/
int ClipLineAgainstHyperplane( Pi, seg, clipped)
Hyperplane Pi;
Segment seg, *clipped;
{
    Scalar d1 = EvalHyperplane(Pi, seg.p1);
    Scalar d2 = EvalHyperplane(Pi, seg.p2);
    Point I;
```

```
    if (d1 > 0 && d2 > 0) {
        /* Both points outside --- trivial reject. */
        return FALSE;
    }
    if (d1 < 0 && d2 < 0) {
        /* Both points inside --- trivial accept. */
        return TRUE;
    }
    if (d1 == 0 && d2 == 0) {
        /* Both points on the boundary --- trivial accept. */
        return TRUE;
    }
    /*
    ** The line intersects the Hyperplane at the point I that breaks seg
    ** into relative ratios abs(d1):abs(d2).
    */
    I = PPrr( seg.p1, seg.p2, fabs( d1), fabs( d2));
    /* Determine which portion of segment to clip away. */
    if (d1 < 0) {
        /* Clip off the segment from I to seg.p2 */
        clipped.p1 = seg.p1;
        clipped.p2 = I;
    } else {
        /* Clip off the segment from seg.p1 to I */
        clipped.p1 = I;
        clipped.p2 = seg.p2;
    }
    return TRUE;
}
```

The coordinate-free implementation of clipping has the added benefit that the code has no notion of the dimension of the space in which the line segments live. This means that the procedure above can be used for 2D as well as 3D line clipping. For 2D clipping, the oriented hyperplanes are the oriented lines that bound the visible window. Notice too that the hyperplanes are not required to be in any special orientation (as long as the clipping volume is convex). This allows irregular windows and clipping volumes to be used without increasing the complexity of the code.

5.3.3. Transformation to Screen Space

The transformation that carries points in the `World` space into points in the `Screen` space should be such that the window on the projection plane is carried into the viewport, as indicated by Figure 19. This transformation, called `ViewTransform`, is an affine map since orthographic projection is assumed (if a perspective view was desired, `ViewTransform` would be a projective map).

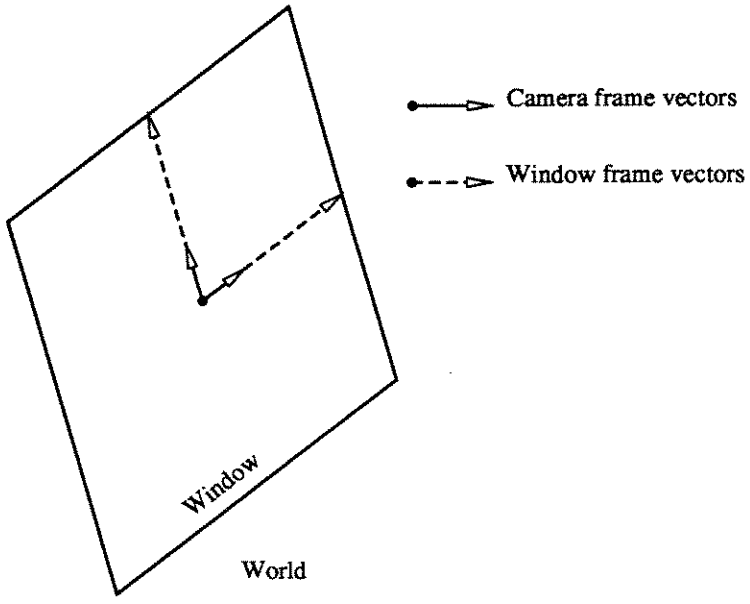

FIGURE 21. The `Window` frame

As described in Section 3.3, an affine transformation such as `ViewTranform` is completely characterized once its action on a frame is known. Referring to Figure 21, a convenient frame for this purpose is obtained from the `Camera` frame by stretching the x and y direction vectors so that they touch the boundaries of the window. We call the frame thus constructed the `Window` frame. The image of the origin of the `Window` frame is `vpcenter`, the center of the viewport; the image of the x-direction vector of the `Window` frame is the vector `vphoriz` that extends horizontally from `vpcenter` to the edge of the viewpoint; the image of the y-direction vector of the `Window` frame is the vector `vpvert` that extends vertically from `vpcenter` to the edge of viewport; finally, since orthographic projection is to be performed,

the z-direction vector of the Window frame is mapped to the zero vector in Screen space. The ViewTransform mapping can therefore be constructed as:

```
BuildViewTransform()
{
    Frame Window;
    Point vpcenter;
    Vector vphoriz, vpvert;

    Window = FCreate( "Window",
                    FOrg( Camera),
                    SVMult( wwidth/2.0, Fx(Camera)),
                    SVMult( wheight/2.0, Fy( Camera),
                    Fz( Camera));
    vpcenter = PCreate( StdFrame(Screen), vpx, vpy);
    vphoriz  = VCreate( StdFrame(Screen), vpwidth/2.0, 0.0);
    vpvert   = VCreate( StdFrame(Screen), vpheight/2.0, 0.0);

    ViewTransform = ACreate( Window, vpcenter, vphoriz, vpvert, VZero(Screen));
}
```

Line segments can then be mapped from the World space into the Screen space by:

```
/*
** Send the segment "seg" through the viewing transformation.
*/
Segment TransformSegment( seg)
Segment seg;
{
    Segment ScreenSegment;

    ScreenSegment.p1 = PAxform( seg.p1, ViewTransform);
    ScreenSegment.p2 = PAxform( seg.p2, ViewTransform);

    return ScreenSegment;
}
```

5.3.4. Scan Conversion

Once the segment has been mapped to Screen space, scan conversion can occur after the device coordinates for the endpoints have been determined. Using the geometric ADT, this is most easily accomplished by defining a "device frame" in the Screen space. This frame is defined such that coordinates relative to the device frame represent *device coordinates.*

As an example, consider a device that has the origin in the upper left hand corner with x coordinates increasing to the right and y increasing downward. Suppose too that pixels are addressed from 0 to $X_{\mathbf{max}}$ in the x direction and from 0 to $Y_{\mathbf{max}}$ in the y direction.

Referring to Figure 19, the wireframe example establishes the convention that the visible portion of the Screen space is the unit square subtended by `StdFrame(Screen)`. Let the origin of `StdFrame(Screen)` be denoted by $\mathcal{O}_s$, the x-direction vector by $\vec{x}_s$, and the y-direction vector by $\vec{y}_s$. If the device frame has origin $\mathcal{O}_d$, x-direction vector $\vec{x}_d$ and y-direction vector $\vec{y}_d$, then:

- $\mathcal{O}_d = \mathcal{O}_s + \vec{y}_s$. This sets the origin of the device frame to the upper left hand corner of the visible region of Screen space.
- $\vec{x}_d = \frac{1}{X_{\mathrm{max}}}\vec{x}_s$. This says that device x-coordinates increase to the right and range from 0 to $X_{\mathbf{max}}$.
- $\vec{y}_d = -\frac{1}{Y_{\mathrm{max}}}\vec{y}_s$. This says that device y-coordinates increase to downward and range from 0 to $Y_{\mathbf{max}}$.

Having established the device frame as an initialization step, a segment in `Screen` space can be scan converted by extracting coordinates relative to the device frame, then invoking a standard line drawer such as Bresenham's algorithm [Foley and Van Dam '82]:

```
BuildDeviceFrame()
{
    Point Os,Od;
    Vector Xs, Ys, Xd, Yd;

    Os = FOrg(StdFrame(Screen));
    Xs = Fx(StdFrame(Screen));
    Ys = Fy(StdFrame(Screen));

    Od = PVAdd( Os, Xs);
    Xd = SVMult( 1/XMAX, Xs);
    Yd = SVMult( -1/YMAX, Ys);

    DeviceFrame = FCreate( Od, Xd, Yd);
}

/*
** Draw a segment on the device by extracting device coordinates
** then calling Bresenham's algorithm.
*/
DrawSeg(seg)
Segment seg;
{
    Scalar x1, y1, x2, y2;

    PCoords( DeviceFrame, seg.p1, &x1, &y1);
    PCoords( DeviceFrame, seg.p2, &x2, &y2);

    Bresenham( (int) x1, (int) y1, (int) x2, (int) y2);
}
```

5.4. Example 2: The Ray-Sphere Intersection Problem

At the heart of every ray tracer is a collection of routines that compute the intersection of a ray with primitive objects. In this section, we develop a procedure for computing the intersection of a ray and a sphere using operations provided by the ADT.

Input to the procedure is a ray, represented as a point B and a direction vector $\vec{v}$, and a sphere, represented as a center point C and a radius r (see Figure 22). Output is a point of

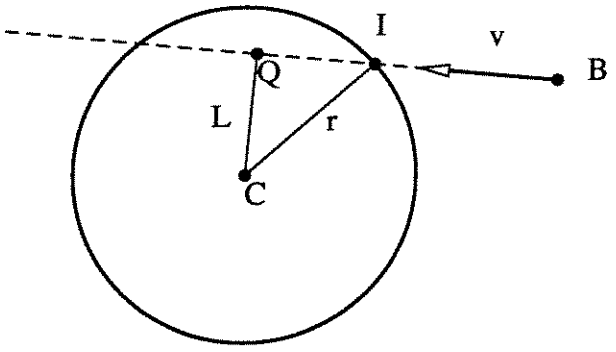

FIGURE 22. Ray-sphere intersection

intersection I and a normal $\vec{n}$ to the sphere at I (the normal is used in the shading calculation and in the casting of reflected and refracted rays).

Assuming that B is outside the sphere, there are two cases when there can be no intersection. The first case occurs when the dot product of $\vec{v}$ and the vector from B to C is negative, signaling the fact that the intersection point is "behind the ray." The second case occurs when the point Q, formed by perpendicularly projecting C onto the ray, is more than one radius away from point C. Point Q is computed by first projecting the vector $C - B$ onto $\vec{v}$. In general, the perpendicular projection of a vector $\vec{u}$ onto a vector $\vec{w}$ is accomplished by (cf. [Goldman '87]):

$$\text{Proj}(\vec{u}, \vec{w}) = \frac{\vec{u} \cdot \vec{w}}{\vec{w} \cdot \vec{w}} \vec{w}.$$

If the squared distance L^2 from Q to C is larger than r^2, then, as mentioned above, there can be no intersection. (Use of the squared distance eliminates an unnecessary square root operation.) If $L^2 < r^2$, then the intersection point I can be computed using the Pythagorean Theorem. Finally, the normal to the sphere at I is the dual of the vector $I - C$. As pseudo-

code:

if$(\vec{v} \cdot (C - B) < 0)$**then**
    return No Intersection
endif
$Q \leftarrow \text{Proj}(C - B, \vec{v}) + B$
$L^2 \leftarrow (Q - C) \cdot (Q - C)$
if $L^2 < r^2$**then**
    $d \leftarrow \sqrt{r^2 - L^2}$
    $I \leftarrow Q - d\vec{v}$
    $\vec{n} \leftarrow \text{Dual}(I - C)$
    return $(I, \vec{n})$
else
    return No Intersection
endif

6. Implementation Notes

To aid implementors of the geometric ADT, we offer a few suggestions taken from our implementation.

Rather than storing frames by separately maintaining the origin and basis vectors, we represent them by two change of coordinate matrices. One of these is the matrix that converts coordinates in this frame into coordinates relative to the pre-defined standard frame of the space. For efficiency in extracting coordinates relative to the frame we also store the inverse of this matrix:

```
typedef struct frame {          /* A coordinate frame.              */
    struct Space *s;            /* The containing space.            */
    char *name;                 /* Printable name for debugging.    */
    Matrix to_stdf;             /* Rep of frame rel to s->stdf.     */
    Matrix from_stdf;           /* Rep of s->stdf rel to frame.     */
} Frame;
```

Spaces are represented as structures that record the dimension of the space, the name of the space (for debugging purposes), and the standard frame (referred to internally as `stdf`):

```
typedef struct Space {          /* A Euclidean space.              */
    int dim;                    /* The dimension of the space      */
    char *name;                 /* Printable name for debugging    */
    Frame stdf;                 /* The predefined Cartesian frame. */
} *Space;
```

Points and vectors are represented by storing their coordinates relative to the standard frame in a row matrix. Normals are represented by storing their coordinates relative to the standard frame in a column matrix. These objects also store pointers to the spaces in which they live. Points, for instance, are represented by the structure:

```
typedef struct {                /* A point                         */
    Space s;                    /* The containing space            */
    Matrix p;                   /* Coordinates rel to s->stdf      */
} Point;
```

When a point (or vector) is created using `PCreate` (or `VCreate`), coordinates relative to the standard frame of the space are immediately computed. More specifically, C-like pseudo-code for the creation of a point in a 3-space `PCreate` is:

```
/*
** Create and return a new point.  The coordinates of the
** point are (c0,c1,c2,1) relative to the frame F.
*/
Point PCreate( F, c0, c1, c2)
Frame F;
Scalar c0, c1, c2;
{
    Point NewPoint;

    /* Record the space in which the point lives */
    NewPoint.s = F.s;
    /*
    ** Compute and store the coordinates of the point relative
    ** to the standard frame.
    */
    NewPoint.p = (c0 c1 c2 1) * F.to_stdf;

    return NewPoint;
}
```

Conversely, when coordinates of a point P are extracted relative to some frame F using PCoords, the coordinates of P relative to the standard frame are multiplied by F.from_stdf to compute the coordinates relative to F.

Finally, affine and projective maps are represented by storing pointers to their domain and range spaces, together with their matrix representation relative to the standard frames in the domain and the range. For efficiency in transforming normals, affine maps also store the inverse matrix:

```
typedef struct {            /* An affine transformation                    */
    Space range, domain;    /* The domain and range spaces                 */
    Matrix t;               /* Rep rel to range->stdf and domain->stdf */
    Matrix invt;            /* Inverse of t                                */
} AffineMap;
```

The basic routine for the creation of affine transformations is ACreate. C-like pseudo-code for its implementation is:

```
/*
** Return the affine transformation that carries the origin of F onto
** Oprime, and the basis in F onto v0prime, v1prime, v2prime.
*/
AffineMap ACreate( F, Oprime, v0prime, v1prime, v2prime)
Frame F;
Point Oprime;
Vector v0prime, v1prime, v2prime;
{
    AffineMap T;            /* The returned transformation    */
    AffineMap Tprime;       /* An auxiliary matrix.           */
    Matrix Tprime;          /* Rows built from primed objects */

    if Oprime, v0prime, v1prime and v2prime aren't from same space {
        SignalError();
        return NullTransformation;
    }
    T.domain = F.s;
    T.range  = Oprime.s;
    Tprime = Matrix whose rows are the standard coordinates of v0prime,
             v1prime, v2prime, and OPrime;

    T.t = F.from_stdf * Tprime;
```

```
    if T.domain and T.range have the same dimension {
        T.invt = Inverse( T.t);
    }

    return T;
}
```

7. Summary

It was shown that traditional coordinate-based approaches to geometric programming lead to programs that are geometrically ambiguous, and potentially geometrically invalid. To combat these deficiencies, a geometric algebra and an associated coordinate-free abstract data type were defined. Programs written using the abstract data type are geometrically unambiguous; moreover, they are guaranteed to be geometrically valid.

We believe that the use of the abstract data type also results in programs that are easier to develop, debug, and maintain. The development and debugging phases are improved due to the high degree of type-checking that is possible. Maintainability is improved because the abstract data type forces the applications programmer to clearly define coordinate systems and geometric spaces. This extra level of specification allows programmers other than the original author to quickly understand the geometric behavior of the code.

Acknowledgements

Any "original" ideas that may be contained in these notes have been derived and honed in the course of innumerable discussions with many people. Greg Nielson was the person who first suggested to me that projective concepts were largely unnecessary in graphics and CAGD. Based on Greg's insight, I began teaching the introductory computer graphics class at the University of Washington using affine concepts. This was made much simpler by Ron Goldman's observation that points and vectors could be represented using matrices with an additional component without adopting projective geometry and homogeneous coordinates.

Charles Loop, Steve Mann, Jamie Painter, Ken Sloan, and Kevin Sullivan are largely responsible for forcing me to realize that I generally didn't know what I was talking about. Without their criticisms I would never have refined the ideas to their present state.

References

[Blinn and Newell '78] James F. Blinn and Martin E. Newell, "Clipping Using Homogeneous Coordinates," pp. 281–287 in *Tutorial: Computer Graphics,* Second Edition, edited by John C. Beatty and Kellogg S. Booth, IEEE Computer Society, 1982 (reprinted from *Proceedings of SIGGRAPH '78*).

[de Boor '87] Carl de Boor, "B-Form Basics," pp. 131–148 in in *Geometric Modeling: Algorithms and New Trends,* edited by Gerald Farin, SIAM, 1987.

[Dodson and Poston '79] Dodson, C. T. J., and Poston, T., *Tensor Geometry: The Geometric Viewpoint and its Uses,* Pitman, London, (paperback edition, 1979).

[Farin '86] Gerald Farin, "Triangular Bernstein - Bézier Patches," *Computer Aided Geometric Design,* Vol. 3, No. 2, August 1986, pp. 83-127.

[Flohr and Raith '74] F. Flohr and F. Raith, "Affine and Euclidean Geometry," pp. 293–383 in *Fundamentals of Mathematics, Volume II: Geometry,* edited by H. Behnke, F. Bachmann, K. Fladt, and H. Kunle, translated from German by S. H. Gould, The MIT Press, 1974.

[Foley and Van Dam '82] J. D. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics,* Addison-Wesley, 1982.

[Goldman '85] Ronald N. Goldman, "Illicit Expression in Vector Algebra," *ACM Transactions on Graphics,* Vol. 4, No. 3, July 1985, pp 223–243.

[Goldman '87] Ronald N. Goldman, "Vector Geometry: A Coordinate-Free Approach," *SIGGRAPH '87 Tutorial Course Notes,* Course No. 19, 1987.

[Kowalsky '71] Hans-Joachim Kowalsky, *Einführung in die lineare Algebra,* Walter de Gruyter, New York, 1971.

[O'Nan '76] Michael O'Nan, *Linear Algebra,* Second Edition, Harcourt Brace Jovanovich, Inc., New York, 1976.

[Riesenfeld '81] Richard F. Riesenfeld, "Homogeneous Coordinates and Projective Planes in Computer Graphics," *IEEE Computer Graphics and Applications,* January 1981, pp. 50–55.

[Spivak '79] Michael Spivak, *A Comprehensive Introduction to Differential Geometry, Vol. I,* Second Edition, Publish or Perish, Boston, 1979.

[Stroustrup '86] Bjarne Stroustrup, *The C++ Programming Language,* Addison-Wesley, 1986.

[Weyl '22] Hermann Weyl, *Space, Time and Matter,* translated from German by H. L. Brose, Methuen & Co., London, 1922.

Appendix 1: A Brief Review of Linear Algebra

A *vector space* over the reals is a set $\mathcal{V}$, each element of which is called a *vector*, that satisfies the following properties:

(i) Addition of vectors and multiplication by real numbers (scalars) is defined.

(ii) The set is closed under linear combinations. That is, if $\vec{v}, \vec{w} \in V$, and $\alpha, \beta \in \Re$, then $\alpha\vec{v} + \beta\vec{w} \in \mathcal{V}$.

(iii) There is a unique *zero vector* $\vec{0} \in \mathcal{V}$, such that

- For every vector $\vec{v} \in \mathcal{V}$, $\vec{0} + \vec{v} = \vec{v}$.
- For every vector $\vec{v} \in \mathcal{V}$, $0 \cdot \vec{v} = \vec{0}$.

Some examples of vector spaces are listed below. For each example, think about how multiplication and addition of vectors is defined.

1. $\Re^2 = \{(x, y) | x, y \in \Re\}$.
2. $P^3 =$ the set of all polynomials of degree ≤ 3.
3. $C([0,1]) =$ the set of all continuous functions defined on the unit interval.

The vectors $\vec{v}_1, ..., \vec{v}_k$ are said to be *linearly independent* if

$$c_1\vec{v}_1 + c_2\vec{v}_2 + \cdots + c_k\vec{v}_k = \vec{0} \Leftrightarrow c_i = 0, \quad i = 1, ..., k$$

where $c_1, ..., c_k$ are scalars. Otherwise, the vectors are said to be linearly dependent.

The *dimension* of a vector space is defined to be the largest number of linearly independent vectors. For example, the dimension of $\Re^2$, written dim $\Re^2$ can be shown to be 2, and dim P^3 can be shown to be 4.

A sequence $(\vec{v}_1, ..., \vec{v}_n)$ of linearly independent vectors in a vector space of dimension n is called a *basis*. As a simple example, a basis for $\Re^2$ is $((1,0),(0,1))$. Another basis for $\Re^2$ is $((1,1),(0,1))$, and a basis for P^3 is the familiar *power basis* $(1, x, x^2, x^3)$. Another familiar basis for P^3 are the cubic *Bernstein polynomials* $(x^3, 3x^2(1-x), 3x(1-x)^2, (1-x)^3)$.

Bases are essential for imposing coordinates on vector spaces. The importance is underscored by the following theorem.

THEOREM: Let $(\vec{v}_1, ..., \vec{v}_n)$ be a basis for a vector space $\mathcal{V}$. For every $\vec{w} \in V$, there exists a unique set of scalars $c_1, ..., c_n$ such that

$$\vec{w} = c_1\vec{v}_1 + c_2\vec{v}_2 + \cdots + c_n\vec{v}_n.$$

The numbers $(c_1, ..., c_n)$ are called *coordinates* of $\vec{w}$ *relative to the basis* $(\vec{v}_1, ..., \vec{v}_n)$.

PROOF: For a rigorous proof, see any standard text in linear algebra such as [O'Nan '76].

□

The two important points about this theorem are: (1) coordinates are always relative to some basis, and (2) relative to a particular basis, the coordinates are unique. It is therefore meaningless to talk about the coordinates of a vector without talking about the basis relative to which the coordinates are taken.

Let $\mathcal{V}, \mathcal{W}$ be vector spaces (in many graphics and CAGD applications $\mathcal{V}$ and $\mathcal{W}$ are the same space). A map $T : \mathcal{V} \mapsto \mathcal{W}$ is a *linear transformation* if for every $\alpha_1, ..., \alpha_k \in \Re$, and for every $\vec{v}_1, ..., \vec{v}_k \in \mathcal{V}$,

$$T(\alpha_1\vec{v}_1 + \alpha_2\vec{v}_2 + \cdots \alpha_k\vec{v}_k) = \alpha_1 T(\vec{v}_1) + \alpha_2 T(\vec{v}_2) + \cdots + \alpha_k T(\vec{v}_k).$$

Exercises:

1. For the vector space $\Re^2$, what are the coordinates of $(4, 3)$ relative to the basis $((1, 1), (0, 1))$?
2. For the vector space P^3, what are the coordinates of $2x$ relative to the basis $(1, x, x^2, x^3)$? What are the coordinates of $2x$ relative to the basis $(x^3, 3x^2(1 - x), 3x(1 - x)^2, (1 - x)^3)$?
3. Show that a linear transformation $T : \mathcal{V} \mapsto \mathcal{W}$ carries the zero vector in $\mathcal{V}$ into the zero vector in $\mathcal{W}$.

Appendix 2: Selected Procedures of the Abstract Data Type

Below we list a representative (but not comprehensive) set of descriptions of procedures of our C implementation of the abstract data type. It should be mentioned up front that our implementation is restricted to Euclidean spaces of two and three dimensions. This restriction allows us to statically allocate storage, thereby avoiding sticky issues of dynamic memory management in C.

Where possible, the following convention is used to name procedures: the first several letters (A, F, N, P, or S) indicate either the types of the arguments expected or the type of the return value. Here, A stands for `AffineMap`. F for `Frame`, N for `Normal`, P for `Point` or `ProjectiveMap`, and S for `Space` or `Scalar`.

```
AffineMap AACompose( T1, T2)
AffineMap T1, T2;
```

Return the composition transform defined by T2(T1(.)). An error is signaled if the range of T1 is not equal to the domain of T2.

```
AffineMap ACreate( f, oprime, v0prime, v1prime [, v2prime])
Frame f;
Point oprime;
Vector v0prime, v1prime, v2prime;
```

This is the lowest level affine transformation creation routine. Returned is the affine transformation that carries the origin of frame `f` onto the point `oprime`, the first basis vector of `f` onto the vector `v0prime`, the second basis vector of `f` onto the vector `v1prime`, (and if `f` belongs to a 3-space) the third basis vector of `f` onto the vector `v2prime`. The domain of the transformation is set to be the space in which `f` belongs, the range is the space in which the primed objects belong. An error is signaled if the primed objects do not belong to a common space. Notice that the dimensions of the two spaces can differ.

```
AffineMap ACreateF( f1, f2)
Frame f1, f2;
```

This routine is only appropriate for creating affine transformations between spaces of a common dimension. Returned is the affine transformation that carries the frame `f1` onto the frame `f2`. An error is signaled if `f1` and `f2` belong to spaces of unequal dimension.

```
AffineMap AIdentity( S)
Space S;
```

Return the identity transformation from space S onto itself. This is useful in conjunction with the routines such as ARotate3D, ATranslate, AScale, and AACompose.

```
AffineMap ARotate3D( P, v, theta)
Point P;
Vector v;
Scalar theta;
```

Return the transformation representing a three dimensional rotation about the axis through point P along vector v. The rotation is by an angle theta (given in radians). Positive theta corresponds to a clockwise rotation as seen by an observer looking at P in the direction of -v.

```
ATranslate( v)
Vector v;
```

Return the transformation representing a translation by the vector v. The vector can reside in either a 2-space or a 3-space.

```
Frame FCreate( name, origin, v0, v1 [, v2])
char *name;
Point origin;
Vector v0, v1, v2;
```

Return a new coordinate frame. The name variable is a character string that is associated with the frame that can be used for debugging purposes. The next three or four arguments specify the origin of the frame and a basis of vectors. If the origin and vectors belong to a Euclidean 2-space, then only the vectors v0 and v1 are required; otherwise all three vectors are expected. The order of the vectors is important, especially for use with the coordinate specification and extraction routines (see, eg, PCoords below).

```
Point FOrg( f)
Frame f;
```

Return the origin of the frame f.

```
Vector Fx( f)
Frame f;
```

Return the x-direction vector of frame f.

```
Vector Fy( f)
Frame f;
```

Return the y-direction vector of frame f.

```
Vector Fz( f)
Frame f;
```

Return the z-direction vector of frame `f`.

```
Space InSpace( obj)
AnyType obj;
```

Return the affine space that the geometric object `obj` "lives in". Here `obj` can be either a point, a vector, a normal, or a frame.

```
Normal NCreate( f, c0, c1 [,c2])
Frame f;
Scalar c0, c1, c2;
```

Return a normal vector whose coordinates relative to `f` are `(c0 c1 [c2])`. The coordinate `c2` is needed only if `f` belongs to a 3-space. This routine has the same affect as creating a primal vector whose coordinates are `(c0 c1 [c2])`, then dualizing it (using `VDual`) to create the corresponding normal.

```
void NCoords( norm, f, c0, c1 [, c2])
Normal norm;
Frame f;
Scalar *c0, *c1, *c2;
```

Set the scalars pointed to by `c0`, `c1` (and `c2` if norm belongs to a 3-space) to the coordinates of norm relative to the frame f. The affect is to dualize `norm` to obtain a vector `V`, then to extract the coordinates of `V` relative to the frame `f`.

```
Vector NDual( norm)
Normal norm;
```

Convert `norm` into a primal vector.

```
Scalar NVApply( norm, vec)
Normal norm;
Vector vec;
```

Apply the linear functional represented by `norm` to the vector `vec`. The effect is equivalent to dualizing `norm` to obtain a vector `nv`, then forming the dot product $< \texttt{nv}, \texttt{vec} >$.

```
Point PCreate( f, c0, c1 [, c2])
Frame f;
Scalar c0, c1, c2;
```

Return a point whose coordinates relative to `f` are `(c0 c1 [c2])`. The coordinate `c2` is needed only if `f` belongs to a 3-space.

```
void PCoords( P, f, c0, c1 [, c2])
Point P;
Frame f;
Scalar *c0, *c1, *c2;
```

Set the scalars pointed to by `c0`, `c1` (and `c2` if `norm` belongs to a 3-space) to the coordinates of point `P` relative to the frame `f`.

```
Point PPac( P1, P2, a1)
Point P1, P2;
Scalar a1;
```

Return the point given by the affine combination $a1$ `P1` $+ (1 - a1)$ `p2`.

```
Vector PPDiff( P1, P2)
Point P1, P2;
```

Return the vector given by `P1` $-$ `P2`, that is, the vector from `P2` to `P1`. An error is signaled if `P1` and `P2` belong to different spaces.

```
Scalar PPDist( P1, P2)
Point P1, P2;
```

Return the distance between the points `P1` and `P2`. An error is signaled if `P1` and `P2` belong to different spaces.

```
Point PPrr( P1, P2, r1, r2)
Point P1, P2;
Scalar r1, r2;
```

Return the point that breaks the line segment `P1 P2` into relative ratio `r1` to `r2`. The ratios `r1` and `r2` do not have to sum to unity. For instance, `PPrr( P1, P2, 1, 2)` returns the point one third of the way from `P1` to `P2`.

```
Point PAxform( p, T)
Point p;
AffineMap T;
```

Return the image of p under the transformation T.

```
Point PVAdd( P, v)
Point P;
Vector v;
```

Return the point obtained by adding the point P to the vector v . An error is signaled if P and v belong to different spaces.

```
Space SCreate( name, dim)
char *name;
int dim;
```

Return a new Euclidean space. The name variable is a character string used primarily for debugging purposes; dim is one of TWOSPACE or THREESPACE, denoting that the new space is to be a Euclidean 2-space or 3-space, respectively. As mentioned in the introduction, the returned space S comes with a pre-defined Cartesian frame denoted by StdFrame(S). If the dimension of the space is three, the standard frame is right-handed.

```
Frame StdFrame( S)
Space S;
```

Return the pre-defined Cartesian frame for the space S.

```
Vector SVMult( s, v)
Scalar s;
Vector v;
```

Return the vector given by s*v; that is, perform multiplication of a scalar and a vector.

```
void VCoords( v, f, c0, c1 [, c2])
Vector v;
Frame f;
Scalar *c0, *c1, *c2;
```

Set the scalars pointed to by c0, c1 (and c2 if v belongs to a 3-space) to the coordinates of vector v relative to the frame f.

```
Vector VCreate( f, c0, c1 [, c2])
Frame f;
Scalar c0, c1, c2;
```

Return a vector whose coordinates relative to f are (c0 c1 [c2]). The coordinate c2 is needed only if f belongs to a 3-space.

```
Normal VDual( v)
Vector v;
```

Return the normal vector (ie, linear functional) dual to v.

```
Scalar VMag( v)
Vector v;
```

Return the magnitude of vector v.

```
Vector VNormalize( v)
Vector v;
```

Return the unit vector in the direction of vector v.

```
Vector VVAdd( v1, v2)
Vector v1, v2;
```

Return the vector sum of vectors v1 and v2. An error is signaled if v1 and v2 belong to different spaces.

```
Vector VVDiff( v1, v2)
Vector v1, v2;
```

Return the vector difference of vectors v1 and v2; that is, the vector given by v1 - v2 is returned. An error is signaled if v1 and v2 belong to different spaces.

```
Vector VVCross( v1, v2)
Vector v1, v2;
```

Return the cross product of v1 and v2; that is, the vector v1 x v2 is returned. An error is signaled if v1 and v2 belong to different spaces.

```
Scalar VVDot( v1, v2)
Vector v1, v2;
```

Return the dot product of v1 and v2. An error is signaled if v1 and v2 belong to different spaces.

```
Vector VAxform( v, T)
Vector v;
AffineMap T;
```

Return the image of v under the transformation T. An error is signaled if v is not in the domain of T.

```
Vector VZero(S)
Space S;
```

Return the zero vector in space S.