On the Algebraic and Geometric Foundations of Computer Graphics

RON GOLDMAN
Rice University

Today's computer graphics is ostensibly based upon insights from projective geometry and computations on homogeneous coordinates. Paradoxically, however, projective spaces and homogeneous coordinates are incompatible with much of the algebra and a good deal of the geometry currently in actual use in computer graphics. To bridge this gulf between theory and practice, Grassmann spaces are proposed here as an alternative to projective spaces. We establish that unlike projective spaces, Grassmann spaces do support all the algebra and geometry needed for contemporary computer graphics. We then go on to explain how to exploit this algebra and geometry for a variety of applications, both old and new, including the graphics pipeline, shading algorithms, texture maps, and overcrown surfaces.

Categories and Subject Descriptors: I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*hierarchy and geometric transformations*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Grassmann space, homogeneous coordinates, mass-points, projective space

1. INTRODUCTION

Projective geometry was first introduced into computer graphics in Roberts [1965]. Here Roberts applies homogeneous coordinates to study "points, lines, conic sections, planes, and quadric surfaces," although he states that their "original purpose was to allow perspective transformations." Roberts explains that: "It {homogeneous coordinates} can also be thought of as the addition of an extra coordinate for each vector, a scale factor, *so that the vector has the same meaning after multiplication by a constant*" {emphasis added}. In other words, homogeneous coordinates represent points in projective space. Roberts reiterates this point in several places throughout his paper.

Blinn [1977], Blinn and Newell [1978], Riesenfeld [1981], and most everyone else in the field of computer graphics all follow Roberts' lead, basing their algorithms and computations on projective spaces and homogeneous coordinates. A few sample quotes will suffice to give the common viewpoint espoused by these authors:

> *Briefly, a point is represented as a four-component vector, usually written as* $[x, y, z, w]$. *Any non-zero multiple of this row vector represents the same point*. [Blinn 1977]
>
> *The space represented by homogeneous coordinates is not, however, Euclidian 3-space. It is, in fact, analogous to a topological shape called a projective plane*. [Blinn and Newell 1978]

Author's address: Department of Computer Science, Rice University, 6100 Main Street, Houston, TX 77005-1892, email: rng@cs.rice.edu.

By juxtaposing a description of homogeneous coordinates as used in computer graphics with a topological description of projective planes, this article has documented their equivalence. [Riesenfeld 1981]

Standard textbooks by Newman and Sproull [1973], Patterson and Penna [1986], and Foley et al. [1990] adopt this same general point of view.

Yet we shall see in this article that projective spaces and homogeneous coordinates are incompatible with much of the algebra and a good deal of the geometry currently in actual use in computer graphics, including the construction of basic parametric curves and surfaces, standard shading algorithms, and conventional texture maps. Graphics practitioners are getting correct results by performing calculations for which they have no sound mathematical foundations. The purpose of this article is to infuse mathematical rigor back into computer graphics by proposing Grassmann spaces as an alternative to projective spaces.

Projective spaces fail to support computer graphics on two levels: theoretical and computational, geometric and algebraic. Theoretically, projective spaces are compact, nonorientable manifolds. Such manifolds are not a good geometric model for the visual world; we do not live in such spaces. Gazing along an unobstructed direction in projective space, you would see the back of your head! This phenomenon plays havoc with clipping algorithms, since in projective space points we ordinarily think of as behind the eye are treated no differently from points in front of the eye [Blinn and Newell 1978]. Orientation—up and down, left and right, front and back—plays a fundamental role in the visual world, but there is no notion of orientation in projective space.

Computationally, projective spaces are not vector spaces. Such spaces are not a good algebraic model for computer computations; one cannot even add points in projective space. But without an algebra for points it impossible to represent many of the standard parametric curves and surfaces of graphical design, such as Bezier and B-spline curves and surfaces, in projective space. With no notion of addition or scalar multiplication, projective spaces also fail to support shading algorithms and texture maps based on linear interpolation. Matrix multiplication still works in projective space, but we cannot say that matrix multiplication distributes through addition because there is no notion of addition for points in projective space. We shall see that both of these basic problems—theoretical and computational, geometric and algebraic—can be overcome by replacing projective space with Grassmann space.

The first part of this article, Section 2, deals mainly with theory; the latter part, Section 3, treats applications. We shall begin by reviewing several different possible ambient spaces—vector spaces, affine spaces, projective spaces, and Grassmann spaces—for computer graphics, focusing most of our attention on the transformations in the graphics pipeline. We shall establish that only Grassmann space supports all the algebra and geometry needed for contemporary computer graphics. We then go on to explain how to exploit this algebra and geometry for a variety of applications, both old and new, in computer graphics, including projective transformations, shading algorithms, texture maps, and overcrown surfaces.

This article is a companion article to Goldman [2000], which focuses on the representation of polynomial and rational curves and surfaces for geometric design. A survey of the four candidate ambient spaces can also be found in Goldman [2000], but there the emphasis is on operational models for these spaces, highlighting their ubiquity and utility in computational science and engineering. The article demonstrates that, in addition to vector spaces, affine spaces are required for the representation of polynomial curves and surfaces. Nevertheless: "*Affine space is flawed in two ways*: *both its algebra and its geometry are incomplete. Grassmann space completes the algebra*; *projective space completes the geometry*." The article then goes on to discuss Grassmann spaces and projective spaces and to explain how both of these spaces are required for representing rational curves and surfaces for geometric design.

Expanding on this previous work, this article focuses on specific applications of Grassmann space in computer graphics. The four ambient spaces are reviewed again, but, in contrast to Goldman [2000], here the emphasis is on the transformations in the graphics pipeline associated with each of these different spaces. This article demonstrates that, in addition to vector spaces, affine spaces are required in order to represent the standard transformations of the graphics pipeline. Nevertheless (see Section 2.2): *there are four problems with affine space*, *that*, *taken together*, *make affine space unacceptable as the ambient space for computer graphics*:

(i) *The algebra of affine space is incomplete*.
(ii) *The topology of affine space is not connected*.
(iii) *The geometry of affine space is inadequate*.
(iv) *Perspective projection is not an affine map*.

This article then goes on to discuss projective spaces and Grassmann spaces as possible alternatives to affine space as the ambient space for computer graphics. In Section 2.3, we show that, despite common belief, projective spaces and homogeneous coordinates do not support all the algebra and geometry needed for contemporary computer graphics. We are left then only with Grassmann space. In Section 2.4, we provide, in addition to the standard physical model discussed in Goldman [2000], a simple geometric model for Grassmann space. We also show for the first time how perspective projection is modeled in Grassmann space. The remainder of the article, Section 3, deals with applications of Grassmann space specific to computer graphics, including the analysis of pseudoperspective, shading algorithms, and texture maps.

A word of caution concerning coordinates before we proceed. Many papers and textbooks on computer graphics express algorithms and procedures solely in terms of coordinate computations. Nevertheless, in abstract discussions, coordinates more often obscure than enlighten. Coordinates are low-level constructs well adapted for computing machines but not very serviceable for conveying high level concepts to fellow human beings. Indeed, the indiscriminate use of coordinates often obscures the *meaning* of the computations. We shall therefore avoid coordinates whenever possible, invoking them only when we discuss transformations in terms of matrices. Generally, we shall prefer instead more abstract, higher level constructs such as scalars, points, and vectors. For those who would like to pursue this high level, coordinate free approach further, a general discussion can be found in Goldman [1985], and a coordinate free style of programming is presented in DeRose [1989].

2. THEORIES OF AMBIENT SPACES FOR COMPUTER GRAPHICS

Here we survey the theory of four different mathematical spaces that arise in the study of computer graphics: vector spaces, affine spaces, projective spaces, and Grassmann spaces. We follow Klein's Erlangen program by emphasizing the transformations associated with each particular space. However, the transformations in the graphics pipeline do not exhaust all the algebra needed in computer graphics, so we pay close attention as well to the types of curves for graphical design that can be constructed within these different ambient spaces.

2.1 Vector Space and Linear Transformations

Linear algebra is the study of vector spaces and linear transformations. Informally, a *vector space* is a collection of objects called *vectors* for which the operations of addition, subtraction, and scalar multiplication are defined. Examples of abstract vector spaces are ubiquitous in computational science

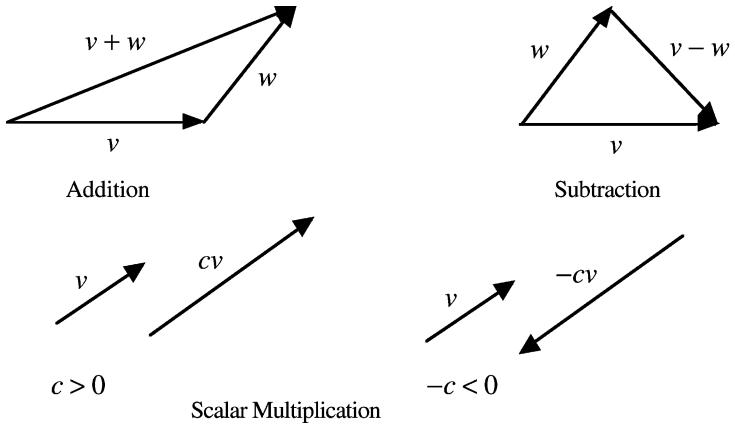


Fig. 1. Addition, subtraction, and scalar multiplication of geometric vectors.

and engineering [Goldman 2000], but here, since we are focusing our attention on computer graphics, we shall concentrate solely on geometric vectors.

In geometry, vectors are represented by arrows. Thus, geometric vectors have both direction and length. These vectors can be added and subtracted by the standard triangle rules, and they can be multiplied by scalars by stretching and shrinking the arrows (see Figure 1).

A *linear transformation* is a map that preserves the operations of addition, subtraction, and scalar multiplication. Thus, a map L is said to be *linear* if

$$L(v \pm w) = L(v) \pm L(w)$$
$$L(cv) = cL(v).$$

In computer graphics, many of the standard transformations, including rotation, reflection, scaling, and orthogonal projection, are linear transformations [Foley et al. 1990].

Often the objects we wish to model in computer graphics lie in 3-dimensions. We can attach three rectangular coordinates to each vector v in 3-space by specifying its components v_1, v_2, v_3, in the x, y, z directions. Since $v = v_1 i + v_2 j + v_3 k$, it follows by linearity that

$$L(v) = L(v_1 i + v_2 j + v_3 k) = v_1 L(i) + v_2 L(j) + v_3 L(k).$$

Overloading our notation—using L to represent both a linear transformation and the corresponding 3×3 matrix

$$L = \begin{pmatrix} L(i) \\ L(j) \\ L(k) \end{pmatrix} = \begin{pmatrix} L_{11} & L_{12} & L_{13} \\ L_{21} & L_{22} & L_{23} \\ L_{31} & L_{32} & L_{33} \end{pmatrix}$$

—it follows that

$$L(v) = v * L = (v_1\ v_2\ v_3) * \begin{pmatrix} L_{11} & L_{12} & L_{13} \\ L_{21} & L_{22} & L_{23} \\ L_{31} & L_{32} & L_{33} \end{pmatrix},$$

where $*$ denotes matrix multiplication. Thus, linear transformations on vectors can be computed by matrix multiplication on their coordinates.

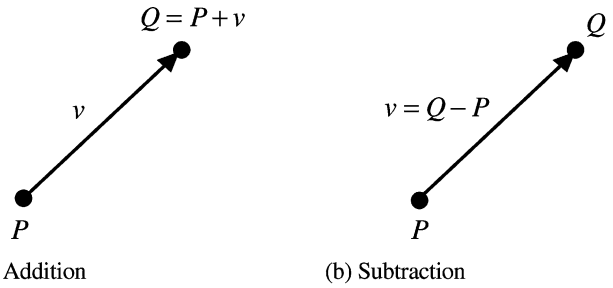


Fig. 2. (a) Addition of a point and a vector and (b) subtraction of a point from a point.

Vectors are very handy for representing geometry. For example, a plane can be represented by a point and a normal vector; an ellipse by its centerpoint together with its semi-major and semi-minor axis vectors. But the essence of computer graphics, what we see on the screen when we look at a picture on a graphics terminal—curves, surfaces and solids—are collections of points not vectors. To incorporate points into our geometry, we need to turn to affine space.

2.2 Affine Space and Affine Transformations

Affine space includes both points and vectors. Geometrically, points are represented by dots, vectors by arrows. Points have position, but not direction or length; vectors have direction and length, but no fixed position. Below we provide a brief description of the geometry and algebra of affine space; for supplementary details and additional models, see Goldman [2000].

Points and vectors can be added and subtracted in the following fashion. To add a vector v to a point P, place the tail of v at P; then the head of v rests at the point $Q = P + v$. Similarly, the difference of two points $Q - P$ is the vector v joining P to Q (see Figure 2). Notice that, by definition, $Q = P + (Q - P)$, so the usual cancellation rules of arithmetic apply.

Addition and scalar multiplication are defined for vectors, but not for points. More generally, we can take arbitrary linear combinations $\sum c_k v_k$ of vectors, but combinations of the form $\sum c_k P_k$ for points seem to have no well-defined, coordinate-free, geometric interpretation. *One of the leitmotifs of this article is the search for meaning for expressions of the form* $\sum c_k P_k$.

Proceeding formally, we can write

$$\sum_{k=0}^{n} c_k P_k = \sum_{k=0}^{n} c_k P_0 + \sum_{k=1}^{n} c_k (P_k - P_0), \tag{2.1}$$

since the terms $c_k P_0, k = 1, \ldots, n$, on the right hand side formally cancel. Now on the right hand side, the term $\sum_{k=1}^{n} c_k (P_k - P_0)$ has a precise meaning, since the expressions $P_k - P_0, k = 1, \ldots, n$, are vectors and linear combinations of vectors are well defined. The problem here is that, in general, the expression

$$\sum_{k=0}^{n} c_k P_0 = \left(\sum_{k=0}^{n} c_k \right) P_0$$

is meaningless, since there is no coordinate free geometric notion of scalar multiplication for points. There are, however, two special cases where scalar multiplication for points makes sense. We can define

$$1 \cdot P = P$$
$$0 \cdot P = 0.$$

Notice that in the second equation, the zero on the left-hand side is the zero scalar, but the zero on the right-hand side is the zero vector. Using these definitions, it follows from Eq. (2.1) that if we want the usual rules for addition, subtraction, and scalar multiplication to hold, then we must define

$$\begin{aligned}\sum_{k=0}^{n} c_k P_k &= P_0 + \sum_{k=1}^{n} c_k (P_k - P_0) \qquad \textit{if} \quad \sum_{k=0}^{n} c_k = 1 \\ &= \sum_{k=1}^{n} c_k (P_k - P_0) \qquad \textit{if} \quad \sum_{k=0}^{n} c_k = 0 \\ &= \textit{undefined} \qquad \textit{otherwise}. \end{aligned} \tag{2.2}$$

Combinations of the form $\sum c_k P_k$ where $\sum c_k = 1$ are called *affine combinations*. Affine combinations provide us with at least a limited algebra for points in affine space. Thus, for example, the midpoint $(P + Q)/2$ between two points P and Q is a well-defined expression in affine space, but $(P + Q)/3$ is a meaningless expression.

For vector spaces, the pertinent maps are linear transformations; for affine spaces, the relevant maps are affine transformations. A transformation A on affine space is said to be an *affine transformation* if A satisfies the following four rules:

(i) A maps points to points.
(ii) A maps vectors to vectors.
(iii) $A(\sum_k c_k v_k) = \sum_k c_k A(v_k)$
(iv) $A(P + v) = A(P) + A(v)$.

The third rule says that A is a linear transformation on the vectors; the third and fourth rules together imply that

(v) $A(\sum_k c_k P_k) = \sum_k c_k A(P_k)$ *whenever* $\sum_k c_k = 1$

—that is, A preserves affine combinations.

Let A be an affine transformation and let us fix a point Q. Since $P = Q + (P - Q)$, it follows by rule iv that

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

Thus, once we know how A acts on the vectors v, we need only know the value of A on a single point Q in order to determine the value of A on any point P.

As we did with vectors, we can attach three rectangular coordinates to each point P in 3-space by fixing an origin O and specifying the components p_1, p_2, p_3 of the vector $P - O$ in the x, y, z directions. However, if we store only three coordinates both for points and for vectors, we will be unable to distinguish the points from the vectors. Therefore, it is common practice to introduce a fourth coordinate, called an *affine coordinate*, to help differentiate points from vectors. The affine coordinate of a point is always 1; the affine coordinate of a vector is always 0—that is,

$$\begin{aligned}(P, 1) &= (p_1, p_2, p_3, 1) \qquad (\textit{point}) \\ (v, 0) &= (v_1, v_2, v_3, 0) \qquad (\textit{vector}).\end{aligned}$$

Notice that the affine coordinate is preserved by affine combinations. Indeed

$$\begin{aligned}\sum_k c_k (P_k, 1) &= \left(\sum_k c_k P_k, 1 \right) \qquad \textit{if} \quad \sum_k c_k = 1 \\ \sum_k c_k (P_k, 1) &= \left(\sum_k c_k P_k, 0 \right) \qquad \textit{if} \quad \sum_k c_k = 0.\end{aligned}$$

Thus, the affine coordinate correctly keeps track of the type of the expression $\sum_k c_k P_k$.

Once we have attached affine coordinates to points and to vectors, we can represent affine transformations A by 4×4 matrices in a manner similar to the way we represented linear transformations by 3×3 matrices. Indeed, since $P = O + p_1 i + p_2 j + p_3 k$ and $v = v_1 i + v_2 j + v_3 k$, it follows by affinity that

$$A(P) = A(O + p_1 i + p_2 j + p_3 k) = A(O) + p_1 A(i) + p_2 A(j) + p_3 A(k)$$
$$A(v) = A(v_1 i + v_2 j + v_3 k) = v_1 A(i) + v_2 A(j) + v_3 A(k).$$

Let L be the linear transformation on vectors induced by the affine transformation A, and let $T = A(O)$. Then substituting into the previous set of equations, we obtain

$$A(P) = T + p_1 L(i) + p_2 L(j) + p_3 L(k)$$
$$A(v) = v_1 L(i) + v_2 L(j) + v_3 L(k).$$

Overloading our notation in the usual manner—using A to represent both an affine transformation and the corresponding 4×4 matrix

$$A = \begin{pmatrix} A(i) & 0 \\ A(j) & 0 \\ A(k) & 0 \\ A(O) & 1 \end{pmatrix} = \begin{pmatrix} L(i) & 0 \\ L(j) & 0 \\ L(k) & 0 \\ T & 1 \end{pmatrix} = \begin{pmatrix} L & 0 \\ T & 1 \end{pmatrix}.$$

—it follows that

$$A(P) = (P, 1) * A = (P - O) * L + T = (p_1 \quad p_2 \quad p_3 \quad 1) * \begin{pmatrix} L & 0 \\ T & 1 \end{pmatrix}$$
$$A(v) = (v, 0) * A = v * L = (v_1 \quad v_2 \quad v_3 \quad 0) * \begin{pmatrix} L & 0 \\ T & 1 \end{pmatrix}.$$

Notice again how the fourth column of A automatically keeps track of the affine coordinates for both points and vectors.

In computer graphics, many of the standard transformations on points and vectors, including rotation, reflection, scaling, orthogonal projection, and translation, are affine transformations [Foley et al. 1990]. Robotics also makes extensive use of affine coordinates and affine transformations to represent rigid motions [Murray et al. 1994]. For example, translation by the vector $T - O$ is represented by the 4×4 matrix

$$A = \begin{pmatrix} I & 0 \\ T & 1 \end{pmatrix},$$

where I is the 3×3 identity matrix. Notice that translation affects points, but has no affect on vectors, which is why we did not encounter translations when we studied vector spaces and linear transformations. Translation is an affine transformation; it is not a linear transformation.

Since affine space contains points as well as vectors, affine space also includes the essential objects of computer graphics: curves, surfaces, and solids. For now, let us fix our attention on parametric curves; similar techniques and observations apply to parametric surfaces.

There are two standard ways to represent polynomial curves in affine space. In the first method, we express a curve $C(t)$ in terms of the monomial basis $1, t, \ldots, t^n$ and write

$$C(t) = C_0 + C_1 t + \cdots + C_n t^n.$$

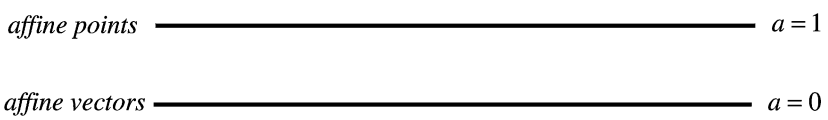


Fig. 3. A schematic view of affine space, containing two disjoint copies of 3-space modeled by the points ($a = 1$) and the vectors ($a = 0$).

Since $C_0 = C(0)$ is a point on the curve, the coefficient C_0 is a point in affine space. But the remaining coefficients $C_1, \ldots, C_n$ must be vectors: since the basis functions $1, t, \ldots, t^n$ do not sum to one, $C_1, \ldots, C_n$ cannot be points. Indeed, $C_1 = C'(0)$ is the tangent vector at $t = 0$. Thus, in the monomial basis, we express a curve as a sum of a point and a linear combination of vectors.

Another polynomial basis commonly used to represent curves in computer graphics is the Bernstein basis [Farin 1997]

$$B_k^n(t) = \binom{n}{k} t^k (1-k)^{n-k} \qquad k = 0, \ldots, n.$$

It is well known that

$$B_0^n(t) + B_1^n(t) + \cdots + B_n^n(t) \equiv 1.$$

Using the Bernstein basis, we represent what is commonly called a *Bezier curve* by setting

$$C(t) = P_0 B_0^n(t) + P_1 B_1^n(t) + \cdots + P_n B_n^n(t).$$

Here the coefficients $P_0, \ldots, P_n$ are indeed points in affine space; the right hand side is well-defined in affine space, since the Bernstein basis functions sum to one.

To apply an affine transformation to a parametric polynomial curve, we need not transform each point on the curve individually; instead, we can simply transform the coefficients. For curves represented in terms of the monomial basis, it follows by affinity that

$$C(t) = C_0 + C_1 t + \cdots + C_n t^n \Rightarrow A(C(t)) = A(C_0) + A(C_1)t + \cdots + A(C_n)t^n.$$

Similarly, for Bezier curves

$$C(t) = P_0 B_0^n(t) + P_1 B_1^n(t) + \cdots + P_n B_n^n(t) \Rightarrow A(C(t)) = A(P_0)B_0^n(t) + A(P_1)B_1^n(t) + \cdots + A(P_n)B_n^n(t).$$

The ability to transform parametric polynomial curves in this simple fashion is an important feature of affine transformations.

Affine spaces have some very nice properties which make them a potentially attractive setting for computer graphics. Affine spaces contain both points and vectors and these entities can be used to represent parametric polynomial curves and surfaces. Moreover, affine transformations include many of the standard transformations of the graphics pipeline, such as translation, rotation, and scaling. Unfortunately, there are four problems with affine space, which taken together make affine space unacceptable as the ambient space for computer graphics.

(1) *The algebra of affine space is incomplete*. It is awkward not to be able to take arbitrary linear combinations of points. Moreover, we introduced affine coordinates to help keep track of the difference between points and vectors. But what is the meaning of these coordinates when the affine coordinate is neither zero nor one?

(2) *The topology of affine space is not connected*. Affine space contains two disconnected components—two disjoint copies of space—modeled by the points and the vectors (see Figure 3). Like its algebra, the topology of affine space calls out for completion.

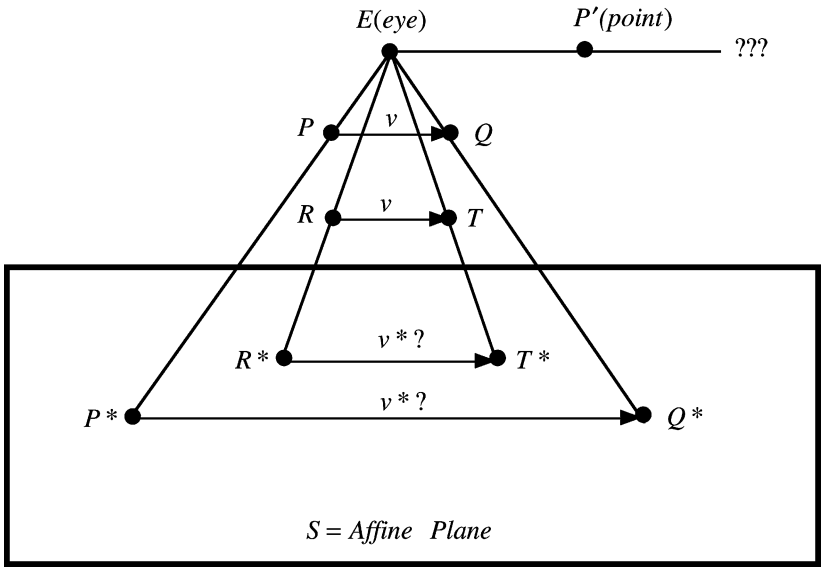


Fig. 4. Perspective projection is not an affine map, since it is not defined for all points or for any vectors.

(3) *The geometry of affine space is inadequate.* Affine space is not rich enough to model all the geometry needed in computer graphics. Parallel lines do not intersect in affine space, though visually such lines seem to intersect. Moreover, rational curves and surfaces do not fit well into the framework of affine space, since the coefficients of rational Bezier curves and surfaces are typically neither points nor vectors [Goldman 2000; 2002] (see also Section 2.4).

(4) *Perspective projection is not an affine map.* Since perspective is fundamental to realistic rendering, we need to adopt a space whose transformations incorporate this essential map.

To see why perspective projection is not an affine map, recall that an affine map must be defined on all points and all vectors. Consider then an affine plane S and a point E outside of the plane S (see Figure 4). To find the perspective projection of a point P from the eye point E to the affine plane S, we must compute the intersection P^* of the line EP with the plane S. But for points P' on the plane through E parallel to S, there is no such affine intersection. Thus, perspective projection is not well defined for every point in affine 3-space. Worse, perspective projection is not defined for any vector in affine 3-space. Consider, for example, the vector $v = Q - P = T - R$ in Figure 4. Recall that for an affine transformation A, we must have

$$A(Q) = A(P + v) = A(P) + A(v)$$
$$A(T) = A(R + v) = A(R) + A(v).$$

Solving for $A(v)$, we find that

$$A(v) = A(Q) - A(P) = A(T) - A(R)$$

or equivalently from Figure 4

$$A(v) = Q^* - P^* = T^* - R^*.$$

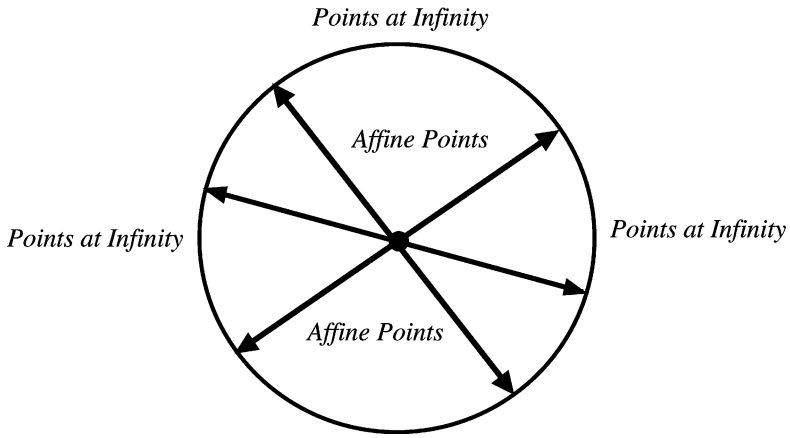


Fig. 5. Projective space consists of affine points and points at infinity. Intuitively, the vectors are pasted on to the affine points as points at infinity, so projective space consists of a single connected component.

But clearly

$$|Q^* - P^*| > |T^* - R^*| \Rightarrow Q^* - P^* \neq T^* - R^*,$$

so perspective projection is not a well-defined affine map on vectors.

To overcome these difficulties, we must seek a new ambient space for computer graphics. Standard textbooks on computer graphics [Foley et al. 1990; Newman and Sproull 1973; Patterson and Penna 1986] commonly suggest projective space, so it is to this ambient space that we now turn our attention.

2.3 Projective Space and Projective Transformations

Projective space consists of two types of points: affine points and points at infinity. The points at infinity solve two problems in computer graphics: they complete the geometry of affine space so that perspective projection can be defined at all points except the center of projection, and they replace the vectors in affine space on which perspective projection is not defined. Notice too that unlike affine space, projective space consists of a single connected component (see Figure 5).

Vectors are commonly used to represent points at infinity, since a point at infinity corresponds to a direction at which parallel lines meet, and a direction can be described by a vector. But vectors incorporate length as well as direction, so the point at infinity represented by the vector v is the same as the point at infinity represented by the vector cv. This difficulty is typically overcome by identifying v and cv with the same point at infinity; that is, nonzero scalar multiples, and even signs, are simply ignored. The zero vector, however, does not represent a point at infinity, since the zero vector does not correspond to any fixed direction.

Points in projective space are represented using homogeneous coordinates, which are an extension of affine coordinates. For vectors, we have seen that we must identify scalar multiples; for points the same convention applies. Thus, in projective space:

$$[v, 0] = [cv, 0] \qquad c \neq 0,$$
$$[P, 1] = [cP, 0] \qquad c \neq 0.$$

The pairs $[cP, c]$, where $c \neq 0$, represent affine points; the pairs $[v, 0]$ represent points at infinity. Now every pair $[X, w] \neq [0, 0]$ has a well-defined meaning in projective space, either as a point in affine

space or as a point at infinity. The parameter w is called the *homogeneous coordinate* [Foley et al. 1990]. We can recover the affine representation of an affine point by dividing its homogeneous representation in projective space by its homogeneous coordinate.

A transformation T on projective space is well defined if

$$T[cv, 0] = c'T[v, 0] \qquad c, c' \neq 0,$$
$$T[cP, 1] = c'T[P, 1] \qquad c, c' \neq 0.$$

Points in projective 3-space are represented using 4 homogeneous coordinates, so every 4×4 matrix represents a well-defined transformation on projective space. Overloading our notation in the usual manner—using T to represent both a 4×4 matrix and the corresponding transformation—for every pair $[X, w] \neq [0, 0]$, we set

$$T[X, w] = (X, w) * T.$$

Since scalar multiplication and matrix multiplication commute:

$$T[cv, 0] = (cv, 0) * T = c(v, 0) * T = cT[v, 0] \qquad c \neq 0$$
$$T[cP, c] = (cP, c) * T = c(P, 1) * T = cT[P, 1] \qquad c \neq 0,$$

so T is indeed a well-defined transformation on projective space. Notice, by the way, that by the same argument T and cT represent the same transformation for any constant $c \neq 0$. We shall be interested here only in transformations on projective space induced by 4×4 matrices; such transformations are called *projective transformations*. In general,

$$T = \begin{pmatrix} t_{11} & t_{12} & t_{13} & t_{14} \\ t_{21} & t_{22} & t_{23} & t_{24} \\ t_{31} & t_{32} & t_{33} & t_{34} \\ t_{41} & t_{42} & t_{43} & t_{44} \end{pmatrix},$$

so a generic projective transformation T need not preserve the distinction between affine points and points at infinity—that is, T may map affine points to points at infinity and points at infinity to affine points. In fact, we shall see shortly that perspective projection behaves precisely in this fashion. In this setting, the affine transformations

$$A = \begin{pmatrix} A(i) & 0 \\ A(j) & 0 \\ A(k) & 0 \\ A(O) & 1 \end{pmatrix}$$

are special, since these transformations are characterized precisely by the property that they preserve the distinction between affine points and points at infinity—that is, the affine transformations are precisely those projective transformations that map affine points to affine points and points at infinity to points at infinity.

To understand how perspective works in projective space, let us now take a careful look at perspective projection. Consider a projective plane S and a projective point E lying outside of the plane S (see Figure 6). To find the perspective projection of an affine point P from the eye point E to the projective plane S, we must compute the intersection P^* of the line EP with the plane S. Similarly, to find the perspective projection of a point at infinity v from the eye point E to the projective plane S, we must compute the intersection v^* of the line Ev with the plane S. To proceed with these computations, we need to recall the equation of a plane in projective space.

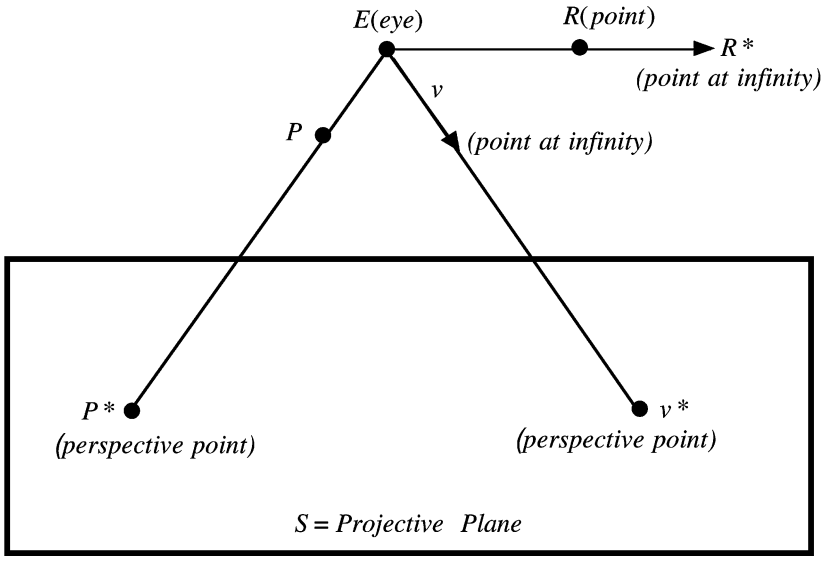


Fig. 6. Perspective projection from an eye point E onto a projective plane S.

A plane in projective space is represented by a homogeneous linear equation

$$Ax + By + Cz + Dw = 0. \tag{2.3}$$

Thus, the four values A, B, C, D completely characterize a plane. Let

$$M = [A, B, C, D].$$
$$X = [x, y, z, w].$$

Then we can rewrite Eq. (2.3) more compactly as

$$M \bullet X = 0, \tag{2.4}$$

where $\bullet$ denotes dot product. Notice that

$$M \bullet X = 0 \Rightarrow M \bullet cX = cM \bullet X = 0.$$

Hence, if X satisfies Eq. (2.4), so does the equivalent projective point cX; moreover, M and cM represent the same plane in projective space.

Now suppose that $M = [A, B, C, D]$ represents the projective plane S. Then, for any projective point X (affine or infinite), the intersection X^* of the line EX with the plane S is given by the projective point

$$X^* = [(X \bullet M)E - (E \bullet M)X]. \tag{2.5}$$

We can verify this formula by observing that X^* lies on the plane S because

$$M \bullet X^* = 0.$$

Moreover X^* lies on the line EX since if M' represents any plane S' containing the points E and X, then

$$M' \bullet E = 0 \quad \textit{and} \quad M' \bullet X = 0 \Rightarrow M' \bullet X^* = 0.$$

Thus, X^* lies on every plane containing the points E and X, so X^* must lie on the line EX. Notice that multiplying any of the points X, M, E by a constant c multiples X^* by c, so the projective point X^* is defined independent of the choice of the homogeneous representatives for X, M, E. The only point X at which X^* is not defined is $X = E$, since, in this case, $X^* = [0, 0, 0, 0]$ is not a point in projective space.

In general, perspective projection maps points at infinity v to affine points

$$v^* = [(v \bullet M)E - (E \bullet M)v].$$

But when the point at infinity v lies in the projective plane S, we have $v \bullet M = 0$, so in this case $v^* = -(E \bullet M)v$ is also a point at infinity; in fact, by the conventions of homogeneous coordinates $v^* = v$. Similarly, perspective projection maps affine points P to affine points

$$P^* = [(P \bullet M)E - (E \bullet M)P].$$

But when P lies on the plane through E parallel to S (i.e., when these two projective planes intersect in the line at infinity), we have $P \bullet M = E \bullet M$, so, in this case, $P^* = (E \bullet M)(E - P)$ is the point at infinity in the direction $E - P$ (see Figure 6).

The 4×4 matrix that represents perspective projection is given by

$$Persp = M^t * E - (E \bullet M)I, \tag{2.6}$$

where I denotes the 4×4 identity matrix, M^t denotes the transpose of M, and $*$ denotes matrix multiplication. Thus,

$$M^t * E = \begin{pmatrix} A \\ B \\ C \\ D \end{pmatrix} * (E_1 \quad E_2 \quad E_3 \quad E_4) = \begin{pmatrix} AE_1 & AE_2 & AE_3 & AE_4 \\ BE_1 & BE_2 & BE_3 & BE_4 \\ CE_1 & CE_2 & CE_3 & CE_4 \\ DE_1 & DE_2 & DE_3 & DE_4 \end{pmatrix}$$

and

$$(E \bullet M)I = \begin{pmatrix} E \bullet M & 0 & 0 & 0 \\ 0 & E \bullet M & 0 & 0 \\ 0 & 0 & E \bullet M & 0 \\ 0 & 0 & 0 & E \bullet M \end{pmatrix}.$$

We can check that the matrix *Persp* does indeed represent perspective projection by observing that

$$X * Persp = X * (M^t * E - (E \bullet M)I) = (X \bullet M)E - (E \bullet M)X = X^*,$$

since $X * M^t = X \bullet M$ and $X * I = X$. Equation (2.6) is quite general and is valid even for eye points E that lie at infinity.

Historically projective space has been touted as the appropriate setting for computer graphics [Patterson and Penna 1986], and indeed projective space has some very compelling properties. Projective space completes the geometry of affine space by including the points at infinity. These points at infinity allow us to intersect lines that are parallel in affine space, and these intersections permit us to extend the definition of perspective projection to all points except the center of projection. The topology of projective space is connected. Projective transformations include all of the standard transformations of the graphics pipeline, such as translation, rotation, scaling, and perspective, and these transformations can all be represented by 4×4 matrices. Point-plane duality as encapsulated in the equation $M \bullet X = 0$ allows us to replace theorems about points in projective space by theorems about projective planes and vice-versa. Moreover, Bezout's theorem, which counts the number of intersections of two algebraic

curves in the plane or three algebraic surfaces in 3-space, is valid only in complex projective space [Cox et al. 1998].

These features of projective space are truly impressive. Nevertheless, there is a heavy price to pay for adopting the conventions of projective geometry, which ultimately makes projective space unwieldy as a vehicle for computer graphics. We discuss the most awkward of these problems below.

(1) *Unnatural Geometry.* We do not live in projective space. If you extend your gaze along an unobstructed straight line in projective space, you will see the back of your head! This phenomenon occurs because straight lines extend right through the point at infinity and come back out on the other side. Indeed, the whole notion of sidedness is foreign to projective geometry. A point does not split a projective line into two disjoint segments as it does in common experience. This phenomenon plays havoc with clipping algorithms, since points we ordinarily think of as behind the eye are treated no differently from points in front of the eye [Blinn and Newell 1978].

(2) *No Vectors.* To introduce the points at infinity in projective space, we needed to relinquish the vectors of affine space. This is indeed a heavy price to pay because it is the vectors that carry information about orientation and distance. Consequently, in projective space there are no notions of orientation and distance. Thus, we see again that the geometry of projective space is not the natural geometry of our visual world where distance and orientation both play decisive roles.

(3) *No Algebra.* Completing the geometry of affine space annihilates its algebra. In affine space, we could at least take affine combinations of points; not so in projective space. Indeed, if P and Q are points in projective space, the value of the expression $(1-\lambda)P + \lambda Q$ depends on the choice of the homogeneous representatives for P and Q. Thus, affine combinations of points are not well-defined points in projective space. As we shall see shortly below, this lack of even a limited algebra for points makes it impossible to represent many of the standard parametric curves and surfaces of graphical design in projective space. Matrix multiplication still works in projective space, but we cannot say that matrix multiplication distributes through addition because there is no notion of addition for points in projective space. Hence, projective transformations are not linear transformations.

(4) *No Standard Parametric Curves and Surfaces.* In affine space, we construct a Bezier curve $C(t)$ by taking an affine combination of control points $P_0, P_1, \ldots, P_n$. That is, we set

$$C(t) = P_0 B_0^n(t) + P_1 B_1^n(t) + \cdots + P_n B_n^n(t).$$

where $B_k^n(t) = \binom{n}{k} t^k (1-t)^{n-k}, k = 0, \ldots, n$, are the standard Bernstein basis functions. But, as we have just seen, affine combinations are not defined in projective space, so we cannot build Bezier curves in projective space. Nor can we adopt projective control points to construct rational Bezier curves, since if we set

$$R(t) = [w_0 P_0, w_0] B_0^n(t) + [w_1 P_1, w_1] B_1^n(t) + \cdots + [w_n P_n, w_n] B_n^n(t),$$

then aside from the rather embarrassing fact that $+$ is not defined on the right-hand side of this equation, we also observe that altering the homogeneous coordinate w_k does not alter the homogeneous point $[w_k P_k, w_k]$; hence, $R(t)$ must also be unchanged, even though changing the weight of a control point is supposed to alter the shape of a rational Bezier curve [Farin 1997].

For all these reasons, projective spaces are not a suitable setting for computer graphics. We shall need to adopt yet another kind of space, retaining the good features of projective spaces, such as the ability to represent perspective projections with 4×4 matrices, without abandoning the algebra and geometry that are required to facilitate computation and representation.

2.4 Grassmann Space

Before embarking on the construction of yet another possible ambient space for Computer Graphics, let us take stock by using our past experience with other geometric spaces to list some criteria that we would like our new space to satisfy.

(1) *Natural Geometry and Topology.* The geometry and topology of our new space should coincide as closely as possible with the geometry and topology of the natural world. There should be no weird optical effects or discontinuities caused by the geometry or topology of the ambient space.

(2) *Points and Vectors.* The space must support both points and vectors. Points are needed to specify positions and vectors are required to specify directions and lengths.

(3) *Complete Algebra.* We should be able to take arbitrary linear combinations of both points and vectors without any annoying restrictions on the coefficients. We also need to incorporate matrix algebra to support the familiar transformations—translation, rotation, scaling, and perspective—of the graphics pipeline.

(4) *Standard Curves and Surfaces.* The ambient space must support the construction of the standard curves and surfaces of graphical design. In particular, we must be able to build Bezier and rational Bezier curves and surfaces inside this space.

This list may seem like a lot to ask, but, in fact, we do not have far to seek for such a space. This time, however, instead of looking to Mathematics, we shall begin by taking our inspiration from classical physics. Later, we shall also provide a simple geometric model for Grassmann space, so we keep the discussion of this physical model brief; for additional details, see Goldman [2000].

In classical mechanics, there are points (locations) and vectors (forces); in addition, there are also objects (masses) on which the forces act. The masses reside at points, so it is often convenient to combine a mass and a point into a single entity called simply a *mass-point*. In this framework, masses are allowed to be negative, so perhaps we should call them charges instead of masses, but the term *mass-point* is fairly standard so we shall stick to it. Assigning mass to points is a old but powerful technique for studying geometry by applying mechanical principles [Kogan 1974; Uspenskii 1961], a method first introduced very early on by Archimedes and refined much later by Grassmann [1894–1911].

Formally, a *mass-point* is a point P in affine space together with a scalar mass $m \neq 0$. Thus it might seem natural to represent a mass-point by the pair (P, m). Unfortunately, the addition rule for mass-points is not very natural in this notation [Goldman 2000], since, as we shall see below, the sum of two mass-points is not simply the sum of the points paired with the sum of the masses. Indeed, as we have seen in Section 2.2, the sum of two points in affine space is not even well defined.

Instead, we shall adopt the notation (mP, m) for the mass-point with mass m located at point P. The expression mP is a convenient but artificial construct, without any natural, coordinate free, physical interpretation. (If we were to introduce rectangular coordinates, then mP would represent the first moment of the mass m around each of the coordinate planes, and (mP, m) would be called the *Grassmann coordinates* of the mass-point.) Formally, we can recover the point P by dividing the expression mP by the mass m. Adopting this convention also allows us to avoid, or at least to postpone, division by storing denominators as masses. Vectors are incorporated into this scheme as entities with zero mass; thus, vectors v are written as $(v, 0)$. This notation for mass-points and vectors is handy because, as we shall see shortly, the sum of two mass-points or a mass-point and a vector can now be computed simply by summing coordinates. We shall see too that scalar and matrix multiplication also work naturally with this notation.

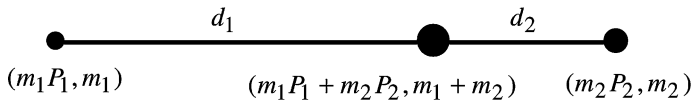


Fig. 7. The sum of two mass-points is located at the center of mass, where $m_1d_1 = m_2d_2$.

To define the sum of two mass-points, we need to specify both the position and the mass of the sum. We define the position to be the center of mass of the two mass-points and the mass to be the sum of the two masses. By Archimedes' Law of the Lever, the center of mass of two mass-points (m_1P_1, m_1) and (m_2P_2, m_2) is located at the point P along the line P_1P_2 characterized by the property that the mass times the distance to the center of mass (first moment) of the first mass-point is the same as the mass times the distance to the center of mass (first moment) for the second mass-point—that is,

$$m_1|P_1 - P| = m_2\,|P_2 - P|.$$

It follows that the center of mass of the mass points (m_1P_1, m_1) and (m_2P_2, m_2) is located at the point $(m_1P_1 + m_2P_2)/(m_1 + m_2)$, since

$$m_1\left|P_1 - \frac{m_1P_1 + m_2P_2}{m_1 + m_2}\right| = \frac{m_1m_2\,|P_1 - P_2|}{m_1 + m_2} = m_2\left|P_2 - \frac{m_1P_1 + m_2P_2}{m_1 + m_2}\right|.$$

Now we see the real convenience of our strange notation for mass-points because

$$(m_1P_1, m_1) + (m_2P_2, m_2) = (m_1P_1 + m_2P_2, m_1 + m_2); \tag{2.7}$$

that is, we can find the center of mass simply by adding coordinates (see Figure 7).

Our inspiration for this definition of addition comes from classical mechanics, where typically we can replace the physical effects of two masses by a single mass that is the sum of the two masses located at their center of mass. Since in this formalism masses can be negative, we also need to worry about what happens when $m_1 + m_2 = 0$. In this case, we define the sum to be the vector from P_1 to P_2 scaled by the mass of P_2. That is,

$$(-mP_1, -m) + (mP_2, m) = (m(P_2 - P_1), 0).$$

To multiply a mass-point (mP, m) by a scalar c, we shall multiply the mass by c and leave the position of the point unchanged. Thus

$$c(mP, m) = (cmP, cm), \tag{2.8}$$

so scalar multiplication can also be performed by multiplying each of the coordinates of the mass-point by the scalar c.

Addition and scalar multiplication are already defined for vectors, so we can just carry over these definitions in the obvious manner. That is, we set

$$(v, 0) + (w, 0) = (v + w, 0)$$
$$c(v, 0) = (cv, 0).$$

To complete our linear algebra of mass-points and vectors, we need to define how to add a vector to a mass-point. Again we take our inspiration from mechanics. Think of the vectors as forces. A force acts on a mass-point by pulling the mass to a new location. But mass has inertia. A force acting on a larger mass will have less of an effect than the same force acting on a smaller mass. A convenient convention is that a force v relocates a mass-point (mP, m) to the new position $P + v/m$. Thus, the larger the mass

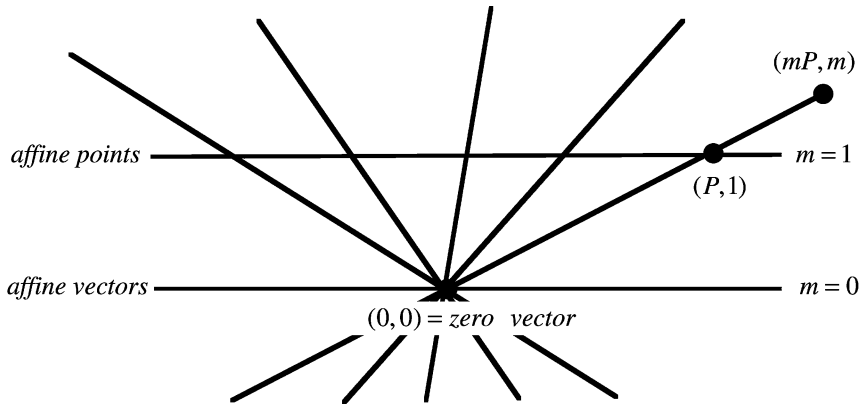


Fig. 8. A geometric interpretation of Grassmann space as the space of all affine vectors, together with the points on the lines connecting the zero vector with the points in affine space.

m, the smaller the effect of the force v. Therefore, we define

$$(mP, m) + (v, 0) = (mP + v, m). \tag{2.9}$$

(We could also think of the vectors v as representing momentum. If a momentum v is imparted to a mass-point (mP, m), then the velocity of the mass-point will be v/m. Hence, in unit time, the mass-point (mP, m) will relocate to $(mP, m) + (v, 0) = (mP + v, m)$.) Thus, once again, to compute the sum, we simply add coordinates. Notice that if a unit mass is located at P, then the force (or momentum) vector v moves the mass-point $(P, 1)$ to the location $P + v$, which is the location of the standard sum of a point and a vector in affine space.

With these definitions of addition, subtraction, and scalar multiplication, the Grassmann space of mass-points and vectors forms a 4-dimensional vector space: 3-dimensions are due to the points in 3-dimensional affine space; the fourth dimension comes from the masses. Since we can take arbitrary linear combinations of mass-points, we have removed the algebraic restrictions inherent in affine space. Moreover, using the notation (mP, m) for mass-points and $(v, 0)$ for vectors, the operations of addition, subtraction, and scalar multiplication are easily accomplished by performing these operations separately on each coordinate.

We have derived Grassmann space from a physical model, but to further our intuitive understanding of Grassmann space, we shall now consider as well a purely geometric model. Recall that affine space consists of two disjoint components represented by the points and the vectors. The points in affine space are indistinguishable, but there is one vector different from all the rest, the zero vector. Grassmann space is the space of all affine vectors, together with the points on the lines connecting the zero vector with the points in affine space (see Figure 8). The point (mP, m), $m \neq 0$, then represents the point on the line $L(t) = (1 - t)(0, 0) + t(P, 1)$—the line from the zero vector through the affine point P—located at the parameter $t = m$. Notice how Grassmann space connects the two disjoint components of affine space by embedding them in a vector space of one higher dimension. Contrast this approach with how the points and vectors are connected together in projective space. Nevertheless, Grassmann space and projective space are intimately related. Each line through the origin in Grassmann space corresponds to a distinct point in projective space. In this way, equivalence classes of points in Grassmann space generate a model for projective space.

We can tie together our physical and geometric models of Grassmann space within a single diagram (see Figure 9). Here we start with two points $(P_1, 1)$, $(P_2, 1)$ in affine space with which we associate the

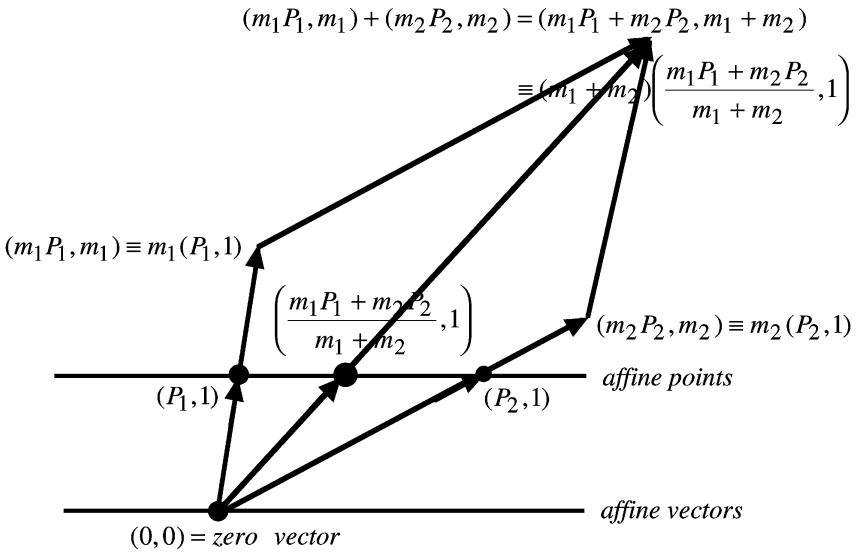


Fig. 9. Two characterizations for addition in Grassmann space. Addition is represented by adding mass-points (dots) in a 3-dimensional affine space or by adding vectors (arrows) in a 4-dimensional vector space.

scalar masses m_1, m_2. As mass-points, their sum is given by the point $((m_1P_1 + m_2P_2)/(m_1 + m_2), 1)$ in affine space associated with the mass $m_1 + m_2$. In this model, mass-points (mP, m) are represented by dots of different sizes in affine space. Alternatively, we can encode a mass-point (mP, m) by the arrow from the zero vector $(0, 0)$ to the affine point $(P, 1)$ scaled by the mass m—that is, we define $(mP, m) = m((P, 1) - (0, 0)) \equiv m(P, 1)$. Now the sum $(m_1P_1, m_1) + (m_2P_2, m_2)$ is given by adding the corresponding arrows $m_1(P_1, 1) + m_2(P_2, 1)$ using the standard triangle rule for addition (see Figure 1). After a bit of algebra, we find that $m_1(P_1, 1) + m_2(P_2, 1) = (m_1 + m_2)((m_1P_1 + m_2P_2)/(m_1 + m_2), 1)$. That is, the sum of the arrows yields the vector $((m_1P_1 + m_2P_2)/(m_1 + m_2), 1)$ scaled by the mass $m_1 + m_2$. Thus, the projection of the arrow $m_1(P_1, 1) + m_2(P_2, 1)$ back into affine space gives the point in affine space corresponding to the addition of the original mass-points (center of mass) and the scale factor is the sum of the original masses. So in the geometric model of Grassmann space, affine points encode direction, mass encodes scale, and these encodings are consistent with the standard addition of mass-points from classical mechanics.

What about transformations? Since Grassmann space is a 4-dimensional vector space, the natural transformations on Grassmann space are linear transformations L represented by 4×4 matrices, which we shall also denote by L. A generic linear transformation L on Grassmann space does not preserve mass since if

$$(x', y', x', w') = (w, y, z, w) * \begin{pmatrix} l_{11} & l_{12} & l_{13} & l_{14} \\ l_{21} & l_{22} & l_{23} & l_{24} \\ l_{31} & l_{32} & l_{33} & l_{34} \\ l_{41} & l_{42} & l_{43} & l_{44} \end{pmatrix},$$

then, in general, $w' \neq w$. Indeed, a generic linear transformation L need not even preserve the distinction between mass-points and vectors—that is, L may map mass-points to vectors and vectors to mass-points—since $w = 0$ does not necessarily imply that $w' = 0$ nor does $w' = 0$ necessarily imply

that $w = 0$. In this setting, the affine transformations

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ a_{41} & a_{42} & a_{43} & 1 \end{pmatrix}$$

are special, since these transformations are characterized precisely by the property that they preserve mass. Hence, affine transformations preserve the distinction between mass-points and vectors.

Perspective projection is not an affine transformation, since perspective is not a well-defined map on vectors. We were able to derive a 4×4 matrix representing perspective projection using the conventions of homogeneous coordinates because in projective space the affine vectors are replaced by points at infinity. But Grassmann space incorporates the vectors. If we are to use Grassmann space to support computer graphics, we need to be able to derive perspective projection using the conventions of mass-points. Let us now see how this works.

A plane in affine 3-space is represented by a linear equation

$$Ax + By + Cz + D = 0. \tag{2.10}$$

As in projective space, the four values A, B, C, D completely characterize a plane. In affine space, these four values have a simple geometric interpretation: the triple $N = (A, B, C)$ is a vector normal to the plane, and the constant $D = -N \bullet Q$, where Q is any point on the plane.

We can multiply Eq. (2.10) by any nonzero scalar w without changing the plane. (Geometrically this amounts to rescaling the normal vector N.) Thus, a mass-point $X = (wx, wy, wz, w)$ in Grassmann space lies on the plane defined by Eq. (2.10) if

$$Awx + Bwy + Cwz + Dw = 0. \tag{2.11}$$

Let

$$\begin{aligned} M &= (A, B, C, D) = (N, -N \bullet Q) \\ X &= (wP, w). \end{aligned}$$

Then we can rewrite Eq. (2.11) more compactly as

$$M \bullet X = wN \bullet (P - Q) = 0. \tag{2.12}$$

Notice how closely Eqs. (2.11) and (2.12) mimic the form of Eqs. (2.3) and (2.4), which represent the equation of a plane in projective space.

Now consider a plane of mass-points S and a mass-point E lying outside of the plane S (see Figure 10). To find the perspective projection of a mass-point P from the eye point E to the plane of mass-points S, we must compute the intersection P^* of the line EP with the plane S. If $M = (A, B, C, D)$ represents the plane S, then

$$P^* = (P \bullet M)E - (E \bullet M)P. \tag{2.13}$$

The proof of Eq. (2.13) is virtually identical to the proof of Eq. (2.5) for the intersection of a line and a plane in projective space! Indeed,

$$M \bullet P^* = 0 \Rightarrow P^* \text{ lie on } S.$$

Moreover, P^* lies on the line EP since if M' represents any plane S' in Grassmann space containing

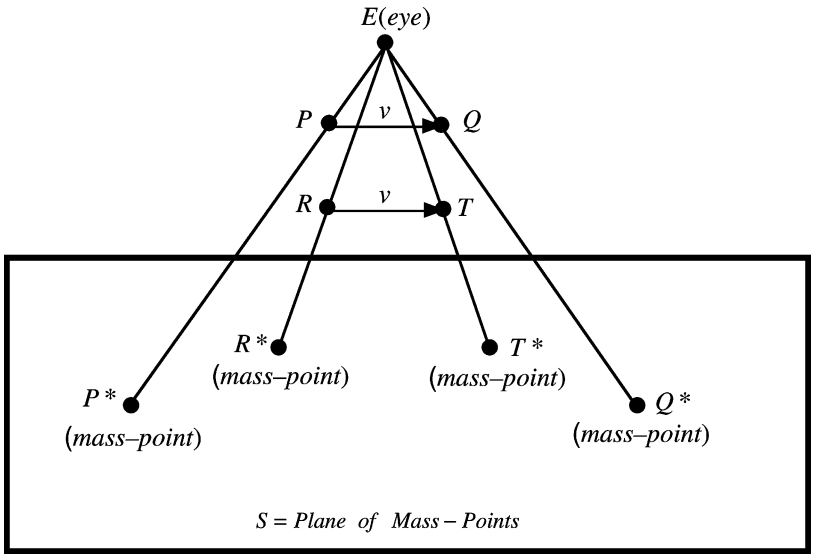


Fig. 10. Perspective projection from an eye point E to the plane of mass-points S.

the points E and P, then

$$M' \bullet E = 0 \quad and \quad M' \bullet P = 0 \Rightarrow M' \bullet P^* = 0,$$

so P^* lies on every plane containing the points E and P.

The 4×4 matrix that represents perspective projection in Grassmann space is also identical to the 4×4 matrix that represents perspective projection in projective space:

$$Persp = M^t * E - (E \bullet M)I. \tag{2.14}$$

Again the verification of this formula is the same as in projective space. Thus, the expressions for perspective are the same in Grassmann space and projective space; only the meaning of the coordinates is changed. This equivalence is, no doubt, in part the reason why people in computer graphics have in the past not thought it necessary to make a clear distinction between Grassmann space and projective space. We shall see shortly, however, that this distinction is vital for many constructions and computations.

Equation (2.14) is quite general and is valid even when the eye point E is replaced by a vector v. Moreover, if E is the vector $v = N/(N \bullet N)$, then *Persp* actually reduces to orthogonal projection onto the plane S.

Perspective projection is not an affine transformation because it is not defined on vectors in affine space. But Grassmann space incorporates these vectors, so somehow perspective projection must be defined on vectors in Grassmann space. How can this be?

Look again at Figure 10. If P, Q, R, T all have unit mass, then

$$v = Q - P = T - R.$$

Therefore, by linearity, we must have

$$Persp(v) = Persp(Q) - Persp(P) = Persp(T) - Persp(R).$$

In affine space $Persp(P), Persp(Q), Persp(R), Persp(T)$ are points and clearly the vectors $Persp(Q) - Persp(P)$ and $Persp(T) - Persp(R)$ are distinct, so $Persp(v)$ is not well defined in affine space. But in

Grassmann space the projections $Persp(P), Persp(Q), Persp(R), Persp(T)$ are mass-points with different masses, so $Persp(Q) - Persp(P)$ and $Persp(T) - Persp(R)$ are also mass-points, not vectors. Moreover, we can verify that $Persp(Q) - Persp(P)$ depends only on v and not on the choice of the points P and Q that represent v, since

$$\begin{aligned} Persp(Q) - Persp(P) &= Q^* - P^* \\ &= \{(Q \bullet M)E - (E \bullet M)Q\} - \{(P \bullet M)E - (E \bullet M)P\} \\ &= (v \bullet N)E - (E \bullet M)v \\ &= Persp(v). \end{aligned}$$

In Grassmann space, perspective projection maps vectors to mass-points, not vectors. This is why perspective projection works in Grassmann space, but not in affine space. We are saved by the masses. In fact, the plane S is not really a plane at all; over each affine point in S is situated a line, each point on the line representing a different mass at the same affine point in the plane. Thus, S extends into 3-dimensions, but the third dimension is mass-like not spatial, which is why we do not see this dimension in Figure 10.

Points and vectors acquire mass under perspective, and these masses can be quite illuminating. For a vector v, the mass of $Persp(v)$ is $v \bullet N$. Thus, only when the vector v is perpendicular to N—that is, when v lies in the plane S—is the mass $v \bullet N = 0$, and $Persp(v) = -(E \bullet M)v$ is a vector not a mass-point. Similarly, if P and E both have unit mass, then the mass of $P^* = Persp(P)$ is $(P - E) \bullet N$. If N is a unit vector, this expression is just the signed distance from the point P to the plane through E parallel to S. In particular, if P lies on the plane through E parallel to S, then $(P - E) \bullet N = 0$ so P^* is a vector not a point. This remark is the analogue of the observation that in projective space perspective projection sometimes maps affine points to points at infinity. This observation is also consistent with the comment that, in general, linear transformations on Grassmann space do not preserve points and vectors. By the way, unlike in projective space, *Persp* is defined at every point in Grassmann space, even at the eye point E, since $Persp(E)$ is just the zero vector. Finally, notice that *Persp* is not the identity on S, even though points on S are not moved by *Persp*. The points on the plane S have mass, and it follows from Eq. (2.13) that the masses of these points are all multiplied by a factor of $-E \bullet M$.

We started this section by listing four criteria that we wanted our ambient space to satisfy. Grassmann space certainly satisfies the first three criteria. The geometry of Grassmann space coincides with the geometry of the physical world. Indeed, we constructed Grassmann space from a physical model. Both points and vectors are included in Grassmann space, and we have a complete algebra since we can take arbitrary linear combinations of mass-points and vectors. Transformations on Grassmann space are represented by 4×4 matrices, so we have indeed incorporated all the familiar transformations of the graphics pipeline.

In Grassmann space, we can also construct the standard curves and surfaces of graphical design. Let $(w_0P_0, w_0), (w_1P_1, w_1), \ldots, (w_nP_n, w_n)$ be a collection of mass-points and let $B_k^n(t) = \binom{n}{k}t^k(1-t)^{n-k}$, $k = 0, \ldots, n$, be the standard Bernstein basis functions. A *rational Bezier curve* is defined by setting

$$R(t) = (w_0P_0, w_0)B_0^n(t) + (w_1P_1, w_1)B_1^n(t) + \cdots + (w_nP_n, w_n)B_n^n(t).$$

Unlike in projective space, this sum makes perfect sense in Grassmann space. Moreover, changing a weight w_k alters the mass of the mass-point (w_kP_k, w_k), thereby altering the shape of the curve. This curve is rational because the affine points $P(t)$ along the curve are computed by dividing the first three coordinates of $R(t)$ by the fourth coordinate; thus

$$P(t) = \frac{\sum_{k=0}^n w_kP_kB_k^n(t)}{\sum_{k=0}^n w_kB_k^n(t)},$$

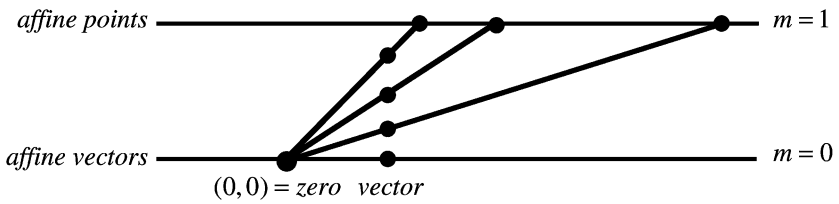


Fig. 11. The natural projection from Grassmann space to affine space is not continuous at the affine vectors. To construct a continuous projection, we need to map the affine vectors to points at infinity—that is, we need to map Grassmann space to projective space.

which is a rational function of t. Finally, when $w_0 = w_1 = \cdots = w_n = 1$, the rational function $P(t)$ reduces to an integral polynomial Bezier curve with affine control points $P_0, P_1, \ldots, P_n$. We shall have more to say about rational Bezier curves and surfaces in Section 3.3.

2.5 Relationships Between the Spaces

We have discussed four distinct kinds of geometric spaces with different types of algebraic structures: vector spaces, affine spaces, projective spaces, and Grassmann spaces. These spaces are intimately related: the affine points live inside of Grassmann space as the points with unit mass and the vectors reside there as well as objects with zero mass, while in projective space all mass-points located at the same affine point but with different mass are identified to the same projective point and vectors are replaced by points at infinity. These observations lead to the following four natural maps between these ambient spaces:

Affine Space $\rightarrow$ *Grassmann Space*
$P \rightarrow (P, 1)$
$v \rightarrow (v, 0)$

Affine Space $\rightarrow$ *Projective Space*
$P \rightarrow [P, 1]$
$v \rightarrow [v, 0]$

Grassmann Space $\rightarrow$ *Affine Space*
$(mP, m) \rightarrow P$
$(v, 0) \rightarrow v$

Grassmann Space $\rightarrow$ *Projective Space*
$(mP, m) \rightarrow [mP, m] = [P, 1]$
$(v, 0) \rightarrow [v, 0] = \left[\frac{v}{|v|}, 0\right]$

Notice that the projections from affine space and Grassmann space to projective space are well-defined everywhere, except at the zero vector.

The natural embedding from affine space to Grassmann space captures the algebraic structure of affine space: both affine combinations and matrix multiplication are preserved. The canonical projection from Grassmann space to affine space is the left-sided inverse of this embedding, but notice that this map is not continuous at the affine vectors (see Figure 11). We shall have more to say about the effects of this discontinuity shortly. The canonical projection from Grassmann space to projective space is continuous everywhere; however, this map preserves only matrix multiplication because linear combinations are not well defined in projective space.

The discontinuity of the projection from Grassmann space to affine space can lead to a problem in the construction of rational Bezier curves and surfaces. Recall that the control structures of rational Bezier curves and surfaces are mass-points. But curves and surfaces in Computer Graphics are collections of points, not mass-points, so to generate graphical curves and surfaces by this construction we need to project from Grassmann space back into affine space by dividing by the mass. Now as long as the mass is not zero—that is, as long as we are not at an affine vector in Grassmann space—this projection yields a well-defined point in affine space. But when the mass is zero, this projection yields a vector in

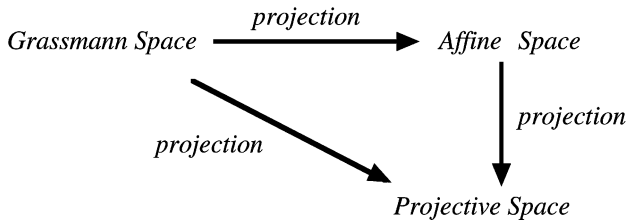


Fig. 12. A commutative diagram of projections. The projection from Grassmann space to affine space is not continuous, even though the continuous projection from Grassmann space to projective space factors through this map.

affine space. Thus, there are really two problems: the curve no longer consists entirely of points, but contains vectors as well, and the curve is not continuous, since the projection from Grassmann space onto affine space is not continuous at the affine vectors. The standard solution to this problem is to replace the projection into affine space by the projection into projective space (see Figure 12). Now objects with zero mass map to points at infinity; moreover, the rational curves we generate are continuous in projective space even at points where the denominator vanishes (see Goldman [2000]). If we insist on continuous curves, then we must adopt projective space to avoid discontinuities; if, however, we are willing to live with discontinuities, which will surely show up anyway on the graphics screen, then the vectors we generate represent asymptotes in affine space and we can dispense entirely with projective space.

The distinctions between Grassmann space and projective space are subtle but important, so let us review them here once again in greater detail. In both spaces, points are represented by pairs $\{mP, m\}$, where P is a point in affine space and m is a nonzero scalar, and in both spaces the affine point P can be recovered from the pair $\{mP, m\}$ by dividing the first three coordinates mP by the fourth coordinate m. But in Grassmann space, mass-points with different masses are distinct; in projective space, homogeneous points that are scalar multiples of one another are the same. Thus, projective space actually consists of equivalence classes of points in Grassmann space. In fact, forming these equivalence classes is one way to construct projective space (see Section 2.4). But addition and subtraction are not so easy to perform on equivalence classes. To make this clear, let us look at an elementary, but revealing, example.

In this simple model, the objects of the Grassmann space are just ordered pairs of integers (p, q); the projective points are equivalence classes of pairs $[p, q]$, where two pairs of integers (p, q) and (r, s) are equivalent if $(mp, mq) = (nr, ns)$ for some nonzero integers m, n. We can identify an ordered pair (p, q) with the fraction p/q and the equivalence class $[p, q]$ with the rational number p/q (or with ∞ if $q = 0$. We are in the habit of identifying a fraction with the corresponding rational number, but a pair (p, q) is not the same as an equivalence class $[p, q]$. Indeed, we can add and subtract fractions by adding and subtracting numerators and denominators

$$\frac{p}{q} \pm \frac{r}{s} = \frac{p \pm r}{q \pm s}, \tag{2.15}$$

but, of course, we cannot add and subtract rational numbers in this manner, since the result would depend on the representatives of the equivalence classes. We can also multiply fractions by integers, using the rule

$$n \times \frac{p}{q} = \underbrace{\frac{p}{q} + \cdots + \frac{p}{q}}_{n} = \frac{np}{nq} \tag{2.16}$$

With these definitions, addition is associative and commutative and scalar multiplication distributes through addition. Various uses of these rules for manipulating fractions, including applications to computer graphics, are presented in Goldman [2001]. Notice that as fractions $p/q \neq np/nq$, even though as rational numbers $p/q = np/nq$. That is, the fractions p/q and np/nq belong to the same equivalence class of rational numbers, but they are not the same fractions, since they do not have the same numerator and denominator. It happens that there are also rules for addition and subtraction of rational numbers,

$$\frac{p}{q} \pm \frac{r}{s} = \frac{ps \pm qr}{qs}, \tag{2.17}$$

but these rules are more complicated than the addition and subtraction rules for fractions, since they must be valid for entire equivalence classes of rational numbers. We can also multiply rational numbers by integers, using the formula

$$n \times \frac{p}{q} = \underbrace{\frac{p}{q} + \cdots + \frac{p}{q}}_{n} = \frac{np}{q}. \tag{2.18}$$

The simple formulas for addition, subtraction, and scalar multiplication of fractions are the rules we use in higher dimensional Grassmann spaces—that is, we implement these operations by applying them to individual coordinates. For example,

$$(m_1P_1, m_1) + (m_2P_1, m_2) = (m_1P_1 + m_1P_2, m_1 + m_2).$$

The rules for rational numbers, however, do not extend to the equivalence classes of higher dimensional projective spaces because these rules are not compatible with the affine points embedded in these spaces. Indeed, the analogue of finding the common denominator of two fractions is to multiply the homogeneous coordinates of two projective points, so in projective space the formula for adding two rational numbers would naturally generalize to

$$[m_1P_1, m_1] + [m_2P_1, m_2] = [m_1m_2P_1 + m_1m_2P_2, m_1m_2] = [P_1 + P_2, 1].$$

This result, however, is inadmissible because $P_1 + P_2$ is not a well-defined point in affine space.

In our simple Grassmann model for fractions, matrix multiplication corresponds to the map

$$\frac{p}{q} * \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \frac{ap + cq}{bp + dq}.$$

Matrix multiplication and scalar multiplication commute because

$$\left(n \times \frac{p}{q}\right) * \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \frac{anp + cnq}{bnp + dnq} = n \times \left(\frac{ap + cq}{bp + dq}\right) = n \times \left(\frac{p}{q} * \begin{pmatrix} a & b \\ c & d \end{pmatrix}\right).$$

Hence, matrix multiplication is also well defined for rational numbers (equivalence classes). Matrix multiplication and scalar multiplication commute as well in higher dimensional Grassmann spaces, which is why the identical matrices appear as transformations in both Grassmann space and projective space. Since the graphics pipeline focuses primarily on matrix computations, people in computer graphics have often been able to get away with performing their analysis in projective space. But these people

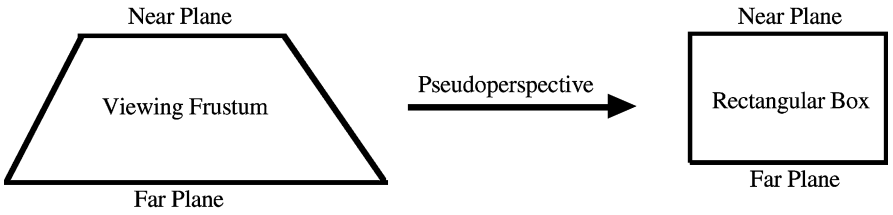


Fig. 13. A schematic view of pseudoperspective—mapping the viewing frustum into a rectangular box.

are bound for trouble analytically as soon as they investigate applications that require any additional vector algebra or where the mass of a mass-point plays a crucial role. We shall now turn our attention to such applications.

3. APPLICATIONS OF GRASSMANN SPACES IN COMPUTER GRAPHICS

Mass-points have many applications. In Euclidian geometry, they can be applied to provide easy proofs for the classical theorems of Ceva and Menelaus [Swimmer 2002]. In computer graphics, mass-points are essential for the definition and analysis of polynomial and rational Bezier curves and surfaces [Fiorot and Jeannin 1989; Ramshaw 2001]. There is also a simple derivation of the formula for perspective projection based solely on Archimedes Law of the Lever [Goldman 2001].

Here we are going to exhibit three additional applications of mass-points in computer graphics: pseudoperspective, shading and texture mapping, and surface overcrown. In each case, we shall see that one of the overriding issues is that in computer graphics *mass matters*. That is, the distinction between coordinates representing the same affine point but with different mass is essential to the derivation.

3.1 Pseudoperspective

In the standard graphics pipeline, there is a key transformation called *pseudoperspective* that maps the viewing frustum into a rectangular box (see Figure 13). This transformation is applied to make polygon clipping simpler and also to prepare the way for certain hidden surface algorithms [Foley et al. 1990; Patterson and Penna 1986]. Replacing the viewing frustum by a box also replaces perspective projection by orthogonal projection, which is typically much simpler to compute. Indeed, if the screen is coincident with the xy-plane, then depth is stored in the z-coordinate and orthogonal projection amounts to just dropping the z-coordinate.

The goals then of pseudoperspective are threefold:

(i) to map the viewing frustum into a rectangular box,

(ii) to replace perspective projection by orthogonal projection, and

(iii) to preserve relative depth for hidden surface algorithms.

How is this done?

The classical way to derive the matrix representing pseudoperspective [Patterson and Penna 1986] is to observe that in 3-dimensions any projective transformation can be defined by specifying the image of 5 generic points. Using some brute force linear algebra—solving 15 homogeneous linear equations in 16 unknowns—we can then construct the 4×4 matrix representing the projective transformation that

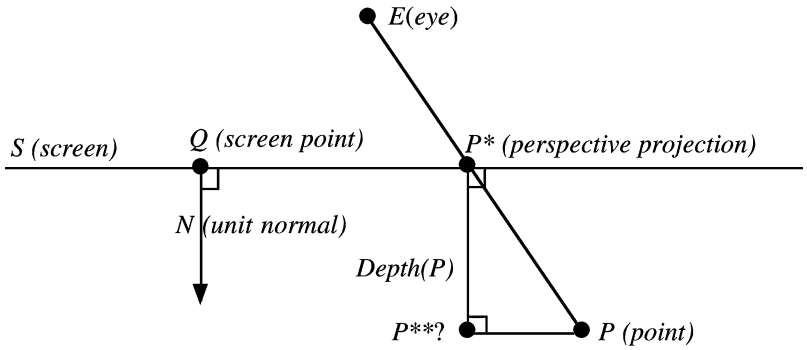


Fig. 14. A schematic view of perspective projection. The affine plane S of the screen is determined by a point Q and a unit normal vector N. The point P^* is the perspective projection of the point P from the eye point E onto the screen S. To map the frustum into the box, the image point P^{**} must lie somewhere along the line through the plane S perpendicular to the point P^*. Here, for simplicity, we have identified the screen with the near plane of Figure 13.

maps a frustum into a box. A computationally easier and conceptually more straightforward approach, which we shall now exhibit, is simply to add the depth vector back into perspective projection.

To fix our notation, suppose that E denotes the eye point, P an arbitrary point in space, and S the plane of the screen determined by a point Q and a unit normal vector N (see Figure 14). A person whose eye is at E will see the point P on the screen S at the position P^*, located at the intersection of the plane S with the line EP. The point P^* can be computed by the perspective projection map derived in Sections 2.3 and 2.4. Indeed by Eqs. (2.5) and (2.13)

$$P^* = (P \bullet M)E - (E \bullet M)P, \tag{3.1}$$

where $M = (N, -N \bullet Q)$, and E and P are also represented using four coordinates—with a 1 in the fourth position to indicate that E and P are indeed affine points, not vectors or points at infinity. (Below we shall sometimes drop the fourth coordinate from E and P as convenience dictates; the representation will be clear from the context.) The distance from the point P to the plane S is given by the projection of the vector $P - Q$ onto the unit normal vector N. Hence

$$Depth(P) = (P - Q) \bullet N = P \bullet M.$$

Thus, to find the point P^{**} lying below the point P^* at the correct depth in the normal direction N, we must compute

$$P^{**} = P^* + \{Depth(P)\}N. \tag{3.2}$$

Let us see now how this computation might work in different ambient spaces.

To perform the computation suggested by Eq. (3.2) in affine space, we must retrieve the affine point corresponding to P^* from Eq. (3.1) by dividing through by the fourth coordinate

$$P \bullet M - E \bullet M = (P - Q) \bullet N - (E - Q) \bullet N = (P - E) \bullet N,$$

so that we can add a point and a vector in affine space. Performing this division yields the point

$$P^* = \frac{[(Q - E) \bullet N]P + [(P - Q) \bullet N]E}{(P - E) \bullet N}.$$

Notice that the coefficients of P and E sum to one, so P^* really is a point in affine space. Indeed, even though the transformation $P \to P^*$ is not an affine transformation, we could have derived this formula

for P^* entirely in affine space by observing that P^* clearly lies on the line PE and P^* must also be on the plane S since $N \bullet (P^* - Q) = 0$. Because the map $P \to P^*$ is not an affine transformation, we cannot hope to derive an affine transformation to generate P^{**}, but perhaps using Eq. (3.2) we can recover a projective transformation. Let us try. Adding a point and a vector in affine space yields:

$$\begin{aligned} P^{**} &= P^* + \{Depth(P)\}N \\ &= \frac{[(Q-E)\bullet N]P + [(P-Q)\bullet N]E}{(P-E)\bullet N} + ((P-Q)\bullet N)N \\ &= \frac{[(Q-E)\bullet N]P + [(P-Q)\bullet N]E + ((P-E)\bullet N)((P-Q)\bullet N)N}{(P-E)\bullet N}. \end{aligned}$$

Unfortunately, the term $((P-E)\bullet N)((P-Q)\bullet N)$ is quadratic in the coordinates of P—each dot product factor is linear—so we cannot represent the transformation $P \to P^{**}$ by a 4×4 matrix. In fact, this formula for P^{**} is not the correct expression for pseudoperspective. Our simple idea of adding the depth vector back into perspective projection fails in affine space.

What about trying projective space? Here there are several problems. First, in projective space there are no vectors, so the normal vector N in Eq. (3.2) has no meaning in projective space. Worse yet, recall that addition is not a well-defined operation on homogeneous coordinates, so the $+$ sign in Eq. (3.2) has no meaning in projective space. Thus, projective space is not an apt setting for implementing Eq. (3.2), which is probably why this approach has not previously been tried.

Our last alternative is to try Grassmann space. In Grassmann space, both the normal vector N and the operator $+$ are well defined. Moreover, we do not need to perform any normalization to do the addition; we need only remember that the fourth coordinate of a vector is zero. Thus, in Grassmann space, we find that

$$P^{**} = P^* + \{Depth(P)\}N = (P \bullet M)E - (E \bullet M)P + (P \bullet M)(N, 0). \tag{3.3}$$

This expression for P^{**} is linear in the coordinates of P, and the corresponding transformation can be represented by the 4×4 matrix

$$Pseudo = M^t * E - (E \bullet M)I + M^t * (N, 0). \tag{3.4}$$

Indeed recalling that

$$P * M^t = P \bullet M$$

it is easy to verify that

$$P * Pseudo = P^{**}$$

Equations (3.3) and (3.4) are the correct formulas for the pseudoperspective transformation [Goldman 1991]. Note how simple, elegant, and straightforward this derivation is compared to the derivation of pseudoperspective using classical projective geometry and linear algebra.

Adding the depth vector back into perspective projection is the intuitive thing to do if we want to retain depth information for hidden surface algorithms. Conceptually, this idea is easy for most students to understand. But there is a catch. The addition we perform is vector addition in Grassmann space, not vector addition in affine space. Recall that when we add a vector to a mass-point in Grassmann space, we do not just translate the point by the vector as in affine space; rather, we must normalize the translation by the mass of the point. Perspective projection introduces mass into points (see Section 2.4). Therefore, when we add the depth vector back into perspective projection, the result is a pseudodepth rather than a true depth due to the normalization of translation by mass. It turns out that this pseudodepth is sufficient for our purposes because all we really need for hidden surface algorithms is

to maintain relative depth, and it is straightforward to verify that relative depth is indeed preserved by pseudoperspective [Goldman 2001].

3.2 Shading Algorithms and Texture Mapping via Linear Interpolation

Points have mass. Does light also have mass?

Let P, $P1$, $P2$ be three points on a polygon and suppose that before perspective or pseudoperspective is performed

$$\begin{aligned} P &= (1-t)P_1 + tP_2 \\ t &= \frac{|P-P_1|}{|P_2-P_1|}. \end{aligned}$$

Then, if we know only the intensities I_1, I_2 at P_1, P_2, we would typically compute the intensity I at P by linear interpolation—that is, we would set

$$I = (1-t)I_1 + tI_2.$$

This formula is essentially Gouraud shading. But that was *before* pseudoperspective. What about *after* pseudoperspective, when, in fact, most shading algorithms are performed? Let P^{**} now denote the point in affine space corresponding to $Pseudo(P)$. Then, it is no longer true that

$$P^{**} = (1-t)P_1^{**} + tP_2^{**}$$

since distance is not preserved by pseudoperspective. In fact,

$$P^{**} = (1-t^*)P_1^{**} + t^*P_2^{**},$$

where

$$t^* = \frac{|P^{**}-P_1^{**}|}{|P_2^{**}-P_1^{**}|} \neq \frac{|P-P_1|}{|P_2-P_1|} = t.$$

How then can we correctly compute shading after pseudoperspective when the original points P_1 and P_2 are no longer available?

Let us take a closer look to see what really goes wrong. Since $Pseudo$ is a linear transformation on Grassmann space (but not on projective space where $+$ is not defined!)

$$Pseudo(P) = (1-t)Pseudo(P_1) + tPseudo(P_2).$$

Expanding this equation in terms of scalar masses and affine points, we obtain

$$(mP^{**}, m) = (1-t)(m_1P_1^{**}, m_1) + t(m_2P_2^{**}, m_2)$$

or equivalently

$$\begin{aligned} mP^{**} &= (1-t)m_1P_1^{**} + tm_2P_2^{**} \\ m &= (1-t)m_1 + tm_2. \end{aligned}$$

So in terms of affine points

$$P^{**} = \frac{mP^{**}}{m} = \frac{(1-t)m_1P_1^{**} + tm_2P_2^{**}}{(1-t)m_1 + tm_2}.$$

Thus, when we express P^{**} via linear interpolation between P_1^{**} and P_2^{**} in affine space, we get

$$\begin{aligned} P^{**} &= (1-t^*)P_1^{**} + t^*P_2^{**} \\ t^* &= \frac{tm_2}{(1-t)m_1 + tm_2}. \end{aligned} \tag{3.5}$$

To compute the correct intensity I at P^{**}, we need to use the original parameter t rather than the new parameter t^* to perform the linear interpolation between the intensities I_1 and I_2. Solving Eq. (3.5) for t in terms of t^*, we obtain

$$t = \frac{t^*m_1}{(1-t^*)m_2 + t^*m_1}. \tag{3.6}$$

Thus, we need only know the masses to retrieve t from t^*.

And that would be the end of the story if it were not for a remarkable innovation first introduced by Heckbert and Moreton [1991] and later rediscovered and publicized by Blinn [1996]. It turns out that there are actually three ways to calculate intensity after pseudoperspective: the Wrong Way, the Right Way, and Blinn's Way. We outline these three techniques briefly below and then discuss each in turn.

The Wrong Way

$$\begin{aligned} P_1 \to (m_1P_1^{**}, m_1) \quad P_2 \to (m_2P_2^{**}, m_2) \\ P^{**} = (1-t^*)P_1^{**} + t^*P_2^{**} = \frac{(1-t)m_1P_1^{**} + tm_2P_2^{**}}{(1-t)m_1 + tm_2} \\ I = (1-t^*)I_1 + t^*I_2 = \frac{(1-t)m_1I_1 + tm_2I_2}{(1-t)m_1 + tm_2} \end{aligned}$$

The Right Way

$$\begin{aligned} P_1 \to (m_1P_1^{**}, m_1) \quad P_2 \to (m_2P_2^{**}, m_2) \\ P^{**} = (1-t^*)P_1^{**} + t^*P_2^{**} \\ t = \frac{t^*m_1}{(1-t^*)m_2 + t^*m_1} \\ I = (1-t)I_1 + tI_2 \end{aligned}$$

Blinn's Way

$$\begin{aligned} P_1 \to (m_1P_1^{**}, m_1) \quad P_2 \to (m_2P_2^{**}, m_2) \\ I_1 \to \left(\frac{I_1}{m_1}, \frac{1}{m_1}\right) \quad I_2 \to \left(\frac{I_2}{m_2}, \frac{1}{m_2}\right) \\ P^{**} = (1-t^*)P_1^{**} + t^*P_2^{**} = \frac{(1-t)m_1P_1^{**} + tm_2P_2^{**}}{(1-t)m_1 + tm_2} \\ I = \frac{(1-t^*)(I_1/m_1) + t^*(I_2/m_2)}{(1-t^*)1/m_1 + t^*(1/m_2)} = (1-t)I_1 + tI_2 \end{aligned}$$

In the first method—commonly used but numerically incorrect—the value t^* is computed from linear interpolation on the points *after* projection. This t^* value is okay for depth calculations where we are only interested in relative depths, but, as we have seen, it is not alright for intensity computations

where we are interested in obtaining absolute, not relative, values. In this method the formulas

$$P^{**} = (1-t^*)P_1^{**} + t^* P_2^{**} = \frac{(1-t)m_1 P_1^{**} + t m_2 P_2^{**}}{(1-t)m_1 + t m_2}$$
$$I = (1-t^*)I_1 + t^* I_2 = \frac{(1-t)m_1 I_1 + t m_2 I_2}{(1-t)m_1 + t m_2}$$

look quite similar. Since $P_1 \to (m_1 P_1^{**}, m_1)$, $P_2 \to (m_2 P_2^{**}, m_2)$, it is as if when we performed pseudoperspective on the points we also mapped $I_1 \to (m_1 I_1, m_1)$, $I_2 \to (m_2 I_2, m_2)$, and then performed linear interpolation on the intensities. Thus, in this method, light inherits mass, just like points—not a good idea after all (but read on).

In the second method—the correct procedure—we compute both t^* and t. The value t^* is computed incrementally in scan-line algorithms and used for depth calculations, whereas t must be computed from t^* for intensity computations. Notice that t^* comes from linear interpolation on the points *after* projection, while t is the correct value for linear interpolation on the points *before* projection. By the way, it is not necessary to use t for depth calculations, since the depth (pseudodepth) is not exact anyway, so the t value is required only for intensity computations. Unlike the previous incorrect method, in this approach light does not inherit mass. Intensity is a scalar and is not affected by pseudoperspective.

What about Blinn's way? In a typical scan-line algorithm, the value of t^* is updated incrementally. Using the second method, we must also compute t from t^*, so there is some additional overhead in using two separate interpolation parameters. A while back Blinn explained an ingenious idea about how to get by with just the single parameter t^* by introducing reciprocal masses into the intensities [Heckbert and Moreton 1991; Blinn 1996]. If we treat the intensities as elements of a Grassmann space and map

$$(I_1, 1) \to \left(\frac{I_1}{m_1}, \frac{I_1}{m_1}\right) \text{ and } (I_2, 1) \to \left(\frac{I_2}{m_2}, \frac{1}{m_2}\right),$$

then the ratio

$$\frac{I}{1} = \frac{I/m}{1/m}$$

defining the intensity is unchanged, so I is still a scalar. But now the intensity corresponding to the linear interpolant

$$(mI, m) = (1-t^*)\left(\frac{I_1}{m_1}, \frac{1}{m_1}\right) + t^*\left(\frac{I_2}{m_2}, \frac{1}{m_2}\right) \tag{3.7}$$

is the correct intensity at P^*, since

$$I = \frac{mI}{m} = \frac{(1-t^*)(I_1/m_1) + t^*(I_2/m_2)}{(1-t^*)1/m_1 + t^*(1/m_2)} = (1-t)I_1 + tI_2.$$

Therefore, in scan-line algorithms, the value of $(I/m, 1/m)$ can be updated incrementally by setting

$$\Delta\left(\frac{I}{m}\right) = \Delta t^*\left(\frac{I_2}{m_2} - \frac{I_1}{m_1}\right)$$
$$\Delta\left(\frac{1}{m}\right) = \Delta t^*\left(\frac{1}{m_2} - \frac{1}{m_1}\right).$$

Notice that the intensities $(I_1/m_1, 1/m_1)$ and $(I_2/m_2, 1/m_2)$ are really elements of a Grassmann space and not elements of a projective space. We cannot replace $(I_1/m_1, 1/m_1)$ by $(c_1 I_1/m_1, c_1/m_1)$ or

$(I_2/m_2, 1/m_2)$ by $(c_2 I_2/m_2, c_2/m_2)$ in Eq. (3.7), since t^* has already been computed from P_1^{**} and P_2^{**}. Indeed, this entire analysis must be carried out in Grassmann space, not in projective space, even though Blinn [1996] claims to be employing homogeneous coordinates. In Blinn's method, light also has mass, which is quite an ingenious use of the power of Grassmann space.

How can this be? Think of it this way. There are actually two transformations: pseudoperspective and division by mass. Pseudoperspective is a linear transformation on Grassmann space and so preserves affine combinations. Since I is a scalar, pseudoperspective does not affect I or its mass. But division is not linear, it's rational; so division does not preserve affine combinations. What must be done then is to divide the mass of I as well. Since I starts out with $mass = 1$, it winds up with $mass = 1/m$. Note that the intensity I does not change; only the mass is changed. I is still a scalar, since.

$$I = \frac{I/m}{1/m}.$$

We have discussed Gouraud shading, but the same observations apply to Phong shading as well. We have only to replace the intensities I with the normal vectors N; the remaining analysis is identical. So either we must interpolate normal vectors by solving for the original interpolation parameter (The Right Way) or we must assign reciprocal weights to these normal vectors and then apply the new interpolation parameter (Blinn's Way). (Typically vectors are affected by projective transformations, but here we do not transform the normal vectors by pseudoperspective because, just like for intensity, we must retain the original data.) In fact, as Blinn points out, the same analysis applies to any method that uses linear interpolation to compute intermediate values, including, in particular, standard texture mapping procedures. Blinn gives some compelling examples to show that The Wrong Way gives really bad looking texture maps that are corrected by applying The Right Way or Blinn's Way [Blinn 1996].

3.3 Overcrown

So far, we have clarified some well-known procedures in computer graphics by appealing to the algebra of Grassmann spaces. But a skeptic may still ask: do Grassmann spaces ever lead to anything new? Here then is a simple, new application based on the algebra of mass-points. Additional novel applications such as morphing can be found in Goldman [2002].

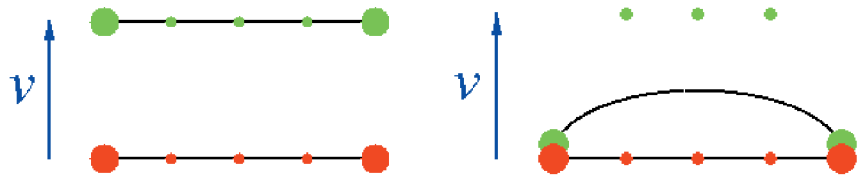
Surfaces fabricated from sheet metal have conspicuous spring back characteristics. Engineers typically compensate for these idiosyncrasies of the material by designing surfaces with bulges or overcrowns, relying on the spring back characteristics of the sheet metal to produce the desired, relatively flat, physical shape. Overcrown is a standard technique in the automotive industry, which relies heavily on sheet metal. We are now going to use the algebra of mass-points to show a simple way to design surfaces with overcrowns.

To simplify the algebra, let us consider first a rational Bezier curve

$$R(t) = \sum_{k=0}^{n} (w_k P_k, w_k) B_k^n(t).$$

In Grassmann space this curve actually consist of two parts: the affine points on the curve

$$P(t) = \frac{\sum_{k=0}^{n} w_k P_k B_k^n(t)}{\sum_{k=0}^{n} w_k B_k^n(t)},$$



(a) $P(t) + v$ (b) $R(t) + (v,0)$

Fig. 15. The effect of adding a vector v to a straight line with large masses at the end points and small masses in the middle: (a) addition in affine space is equivalent to ordinary translation, but (b) mass-point addition in Grassmann space is equivalent to a weighted translation, so the resulting curve is no longer flat but bulges towards the center.

and a mass distribution

$$w(t) = \sum_{k=0}^{n} w_k B_k^n(t).$$

Thus, there is a mass $w(t)$ associated with each curve point $P(t)$.

What happens when we add a vector to a rational Bezier curve? Recall from Section 2.4, Eq. (2.9), that adding a vector to a mass-point translates the point by an amount inversely proportional to the mass of the point. Thus, mass-point addition of a vector v to a rational curve $R(t)$ translates each point $P(t)$ on the curve by a different amount in the direction v because each point on the curve has a different mass $w(t)$. This differential translation is induced by translating each control point by a different amount depending on its weight, sending $P_k \to P_k + v/w_k$. To see this explicitly, simply observe that

$$R(t) + (v, 0) = \sum_{k=0}^{n} (w_k P_k, w_k) B_k^n(t) + \left(\sum_{k=0}^{n} B_k^n(t) v, 0 \right) = \sum_{k=0}^{n} (w_k P_k + v, w_k) B_k^n(t),$$

since

$$\sum_{k=0}^{n} B_k^n(t) \equiv 1.$$

Now suppose we start with a collection of mass-points, each with unit mass, lying along a straight line. Then, the rational Bezier curve $R(t)$ generated by these control points is a straight line with a unit mass at every point. Moreover, if we add a force vector v to this curve, then we simply translate the line $R(t)$ by the vector v, since the mass at every point on the curve is one. Suppose, however, that now we place large masses at the first and last control points of $R(t)$. Since there are large masses at the end points of the curve, these end points hardly move at all when we add the vector v, while the mass-points in the middle of the curve still feel the influence of the force (or momentum) v. The effect is a bulge in the middle of the curve, so the transformed curve is no longer flat (see Figure 15).

We can do much the same for rational Bezier surfaces. Figure 16(a) illustrates a planar rational Bezier surface with a unit mass at each control point. In Figure 16(b), we clamp down the corner points of the surface by placing large masses at the corner control points; in Figure 16(c) we clamp down the edges of the surface by placing large masses at the edge control points.

The geometric effect of adding a force vector to these clamped surfaces is to produce a bulge in the center of the surface, while the surface remains relatively flat and unmoved either at the corners

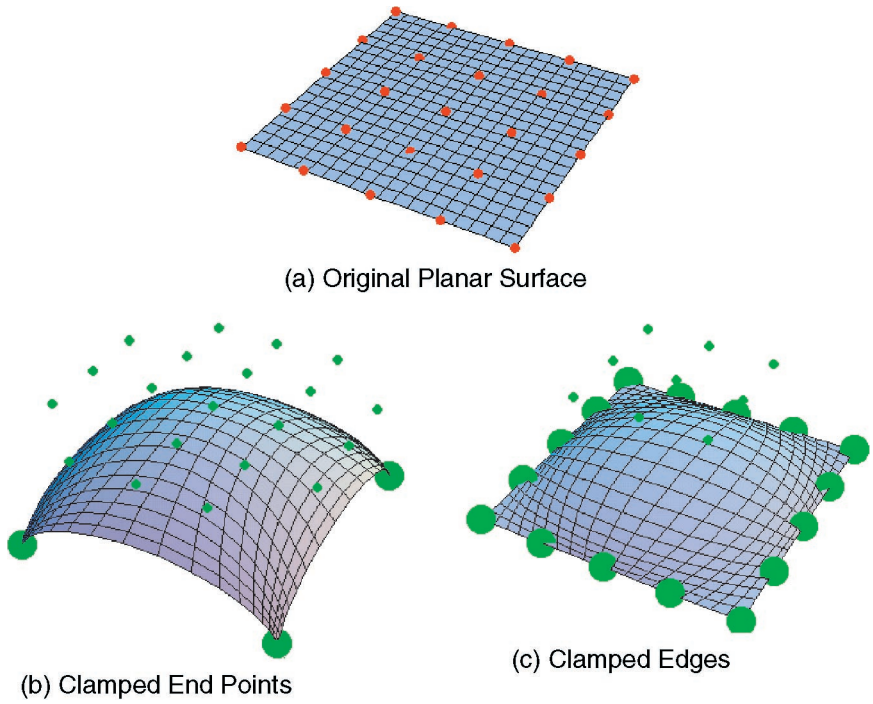


Fig. 16. Overcrown surfaces created by adding a force vector to (a) a planar Bezier patch. In (b) large masses are used to clamp down the corners of the patch, while in (c) large masses are applied to clamp down the edges of the patch.

or along the edges of the patch. Thus, applying mass-point addition of vectors to flat rational Bezier surfaces with cleverly distributed weights is one easy way to design overcrown patches.

4. CONCLUSION

Do we really need to change our ways? Homogeneous coordinates and projective spaces have yielded correct solutions to problems in computer graphics for close to forty years. Is it not arrogant to suggest to the graphics community that they need to abandon well-established methods in favor of novel techniques?

Yes and no. As long as we consider only the graphics pipeline and projective transformations, projective spaces suffice. Early research in computer graphics necessarily focused on properties of the graphics pipeline, so the authors we quoted in the introduction should not be blamed for adopting projective geometry as their geometric paradigm. Indeed, at the time, this choice was quite natural given the focus on visual realism and perspective projection.

But once we step outside the graphics pipeline, the algebra of projective space is inadequate. These algebraic limitations are glaringly evident, for example, in graphical design during the construction of rational Bezier curves and surfaces, when vector spaces are clearly required. This anomaly establishes a tension between the graphics pipeline and graphical design. To accommodate both activities within a single mathematical framework will necessitate a paradigm shift in computer graphics from projective geometry to Grassmann geometry.

Much will be gained. Grassmann space is a vector space; projective space is not. Grassmann space supports a complete linear algebra; projective space only a limited matrix algebra. Contrived projective

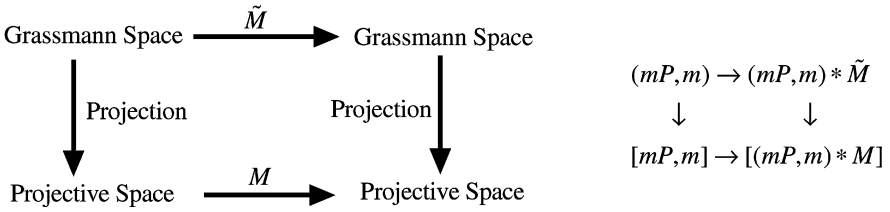


Fig. 17. A projective transformation M on projective space is induced by a linear transformation $\tilde{M}$ on Grassmann space.

transformations on projective space correspond to elementary linear transformations on Grassmann space. Grassmann space incorporates vectors; projective space does not. Grassmann space can model direction and length; projective space cannot. The homogeneous coordinates of projective geometry communicate no visual information; the masses of Grassmann geometry express salient physical features. Projective geometry captures the mathematical intuitions of the visual arts, but Grassmann geometry embodies the physical insights of classical mechanics. The geometry of Grassmann space corresponds closely to the geometry of the physical world; the geometry of projective space does not correlate well with the geometry of the visual world. Thus, Grassmann space gives us access to more algebra than projective space, but with an equally powerful, if not more potent, geometry.

It was only a historical contingency and not a mathematical necessity that projective space and not Grassmann space was chosen by the founders of the field to model the geometry of computer graphics. It is now past time to correct this historical accident. Computer graphics has nothing to lose geometrically, and a great deal to gain algebraically, from adopting the mathematical framework of Grassmann space. Even the graphics pipeline is simpler to analyze in Grassmann space than in projective space (see Section 3.1).

Fortunately for the graphics community, much of the new formalism of Grassmann space is similar to the familiar formalism of projective space. Both spaces introduce an additional coordinate to help keep track of their geometry and both spaces generate the same 4×4 transformation matrices for the graphics pipeline. This equivalence of the transformation matrices is not at all fortuitous, since the two spaces are so closely related (see Section 2.5). Indeed, the *definition* of a projective transformation M on projective space is a map that is induced by a linear transformation $\tilde{M}$ on Grassmann space—that is, for every projective transformation M on projective space there exists a linear transformation $\tilde{M}$ on Grassmann space such that the diagram in Figure 17 commutes.

Hence, the matrices representing the transformations M and $\tilde{M}$ are identical. But the derivation and analysis of $\tilde{M}$ is often algebraically much simpler and more straightforward than the derivation and analysis of M because Grassmann space is a vector space and projective space is not.

Nevertheless, from these observations it follows that many of the formulas, even most of the computations, are identical in both frameworks. The good news then is that programs and code need not change; only their high-level interpretation is different. So there is not that much new to learn, but as we have seen there is much to gain. In fact, it is the thesis of this paper that graphics practitioners have been using Grassmann spaces all along without being aware that they are doing so (see, e.g., Section 3.2). All they need to do now is to consciously abandon the equivalence classes representing points in projective spaces in favor of the autonomous mass-points of Grassmann spaces.

If we append just three words, we can make the statement we quoted from Blinn in the introduction correct:

Briefly, a point is represented as a four-component vector, usually written as [x, y, z, w]. Any nonzero multiple of this row vector represents the same point *in affine space*.

This change is not that much to ask. The result will be an end to the current lack of rigor. There will also be a positive payoff; for replacing *ad hoc* computation with a firm mathematical foundation will provide as well, as demonstrated in Section 3, a broader suite of new techniques and novel applications for computer graphics.

5. ACKNOWLEDGMENT

I would like to thank Al Swimmer for many useful discussions concerning the importance and utility of Grassmann spaces both in theoretical Mathematics and in Computational Science and Engineering. Many of the concepts and insights aired in this article were expressed early on in some of his unpublished manuscripts [Swimmer (2002)].

REFERENCES

BLINN, J. 1977. A homogeneous formulation of a line in 3-space. In *Proceedings of the ACM Conference on Computer Graphics* (*SIGGRAPH '77*). ACM, New York, pp. 237–241.

BLINN, J. 1996. Hyperbolic interpolation. In *Jim Blinn's Corner*: *A Trip Down the Graphics Pipeline*. Morgan-Kaufmann, San Francisco, Calif., pp. 167–178.

BLINN, J. AND NEWELL, M. 1978. Clipping using homogeneous coordinates. In *Proceedings of the ACM Conference on Computer Graphics* (*SIGGRAPH '78*). ACM, New York, pp. 245–251.

COX, D., LITTLE, J., AND O'SHEA, D. 1998. *Using Algebraic Geometry*. Springer-Verlag, New York.

DEROSE, T. D. 1989. A coordinate-free approach to geometric programming. In *Theory and Practice of Geometric Modeling*, W. Strasser and H. P. Seidel, Eds. Springer-Verlag, Berlin, Germany, pp. 291–305.

FARIN, G. 1997. *Curves and Surfaces for Computer Aided Geometric Design*: *A Practical Guide*, fourth edition. Academic Press, San Diego, Calif.

FIOROT, J. AND JEANNIN, P. 1989. *Courbes et Surfaces Rationnelles. Applications à la CAO*. RMA12, Masson, Paris, France. (English version: *Rational Curves and Surfaces. Applications to CAD*. Wiley and Sons, Chichester, England, 1992.)

FOLEY, J. D., VAN DAM, A., FEINER, S. K., AND HUGHES, J. F. 1990. *Computer Graphics*: *Principles and Practice*, second edition. Addison-Wesley, New York.

GOLDMAN, R. N. 1985. Illicit expressions in vector algebra. *ACM Trans. Graph. 4*, 3, 223–243.

GOLDMAN, R. N. 1991. More matrices and transformations: Shear and pseudoperspective. In *Graphics Gems II*, Jim Arvo, Ed. Academic Press, Orlando, Fla., pp. 338–341.

GOLDMAN, R. N. 2000. The ambient spaces of computer graphics and geometric modeling. *IEEE CG&A 20*, 76–84.

GOLDMAN, R. N. 2001. Baseball arithmetic and the laws of pseudoperspective. *IEEE Comput. Graph. Appl. 21*, 70–78.

GOLDMAN, R. N. 2002. Applications of the mass distributions and vector spaces associated with rational Bezier curves and surfaces. In preparation.

GRASSMANN, H. G. 1894–1911. *Gesammelte Mathematische und Physikalische Werke.* B. G. Teubner, Leipzig.

HECKBERT, P. AND MORETON, H. 1991. Interpolation for Polygon Texture Mapping and Shading. In *State of the Art in Computer Graphics*: *Visualization and Modeling*, D. Rogers and R. Earnshaw, Ed. Springer-Verlag, New York.

KOGAN, B. Y. 1974. *The Application of Mechanics to Geometry*. The University of Chicago Press, Chicago, Ill.

MURRAY, R., LI, Z., AND SASTRY, S. 1994. *A Mathematical Introduction to Robotic Manipulation*. CRC Press, Boca Raton, Fla.

NEWMAN, W. AND SPROULL, R. 1973. *Principles of Interactive Computer Graphics*. McGraw-Hill, New York.

PATTERSON, R. AND PENNA, M. 1986. *Projective Geometry and Its Applications in Computer Graphics*. Prentice-Hall, Englewood Califfs, N.J.

RAMSHAW, L. 2001. *On Multiplying Points*: *The Paired Algebras of Forms and Sites*. SRC Research Report #169, COMPAQ Corporation, Palo Alto, Calif.

RIESENFELD, R. 1981. Homogeneous coordinates and the projective plane in computer graphics. *IEEE CG&A 1*, 50–55.

ROBERTS, L. G. 1965. Homogeneous matrix representation and manipulation of N-dimensional constructs. *The Computer Display Review*, C. W. Adams Associates, Inc., Cambridge, Mass., pp. 1–16.

SWIMMER, A. 2002. Ceva, Menelaus and the associative law. Unpublished manuscript.

USPENSKII, V. A. 1961. *Some applications of mechanics to mathematics*. Blaisdell Publishing Company, New York.

Received January 2001; revised October 2001; accepted November 2001