CMSC 427: Lecture 6
More on Geometry and Geometric Programming

Bases, Vectors, and Coordinates: Last time we presented the basic elements of affine and Euclidean geometry: points, vectors, and operations such as affine combinations. However, as of yet we have no mechanism for defining these objects. Today we consider the lower level issues of how these objects are represented using coordinate frames and homogeneous coordinates.

The first question is how to represent points and vectors in affine space. We will begin by recalling how to do this in linear algebra, and generalize from there. We know from linear algebra that if we have 2-linearly independent vectors, $\vec{u}_0$ and $\vec{u}_1$ in 2-space, then we can represent any other vector in 2-space uniquely as a *linear combination* of these two vectors (see Fig. **??**(a)):

$$\vec{v} = \alpha_0\vec{u}_0 + \alpha_1\vec{u}_1,$$

for some choice of scalars α_0, α_1. Thus, given any such vectors, we can use them to represent any vector in terms of a triple of scalars (α_0, α_1). In general d linearly independent vectors in dimension d is called a *basis*. The most convenient basis to work with consists of two vectors, each of unit length, that are orthogonal to each other. Such a collection of vectors is said to be *orthonormal*. The *standard basis* consisting of the x- and y-unit vectors is orthonormal (see Fig. **??**(b)).

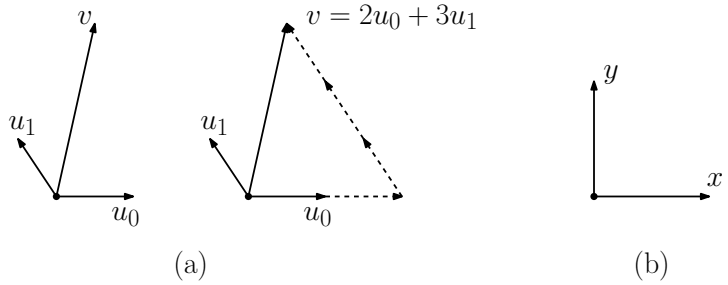


Fig. 1: Bases and linear combinations in linear algebra (a) and the standard basis (b).

Note that we are using the term "vector" in two different senses here, one as a geometric entity and the other as a sequence of numbers, given in the form of a row or column. The first is the object of interest (i.e., the abstract data type, in computer science terminology), and the latter is a representation. As is common in object oriented programming, we should "think" in terms of the abstract object, even though in our programming we will have to get dirty and work with the representation itself.

Coordinate Frames and Coordinates: Now let us turn from linear algebra to affine geometry. To define a coordinate frame for an affine space we would like to find some way to represent any object (point or vector) as a sequence of scalars. Thus, it seems natural to generalize the notion of a basis in linear algebra to define a basis in affine space. Note that free vectors alone are not enough to define a point (since we cannot define a point by any combination of vector operations). To specify position, we will designate an arbitrary a point, denoted o, to serve as the *origin* of our coordinate frame. Observe that for any point p, $p - o$ is just some vector $\vec{v}$. Such a vector can be expressed uniquely as a linear combination of basis vectors. Thus, given the origin point o and any set of basis vectors $\vec{u}_i$, any point p can be expressed uniquely as a sum of o and some linear combination of the basis vectors:

$$p = \alpha_0\vec{u}_0 + \alpha_1\vec{u}_1 + \alpha_2\vec{u}_2 + o,$$

for some sequence of scalars α_0, α_1, α_2. This is how we will define a coordinate frame for affine spaces. In general we have:

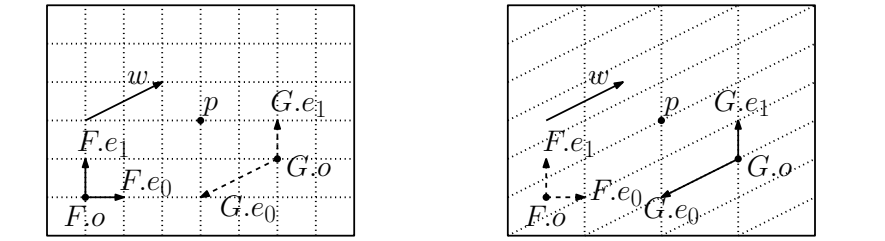
Definition: A *coordinate frame* for a d-dimensional affine space consists of a point, called the *origin* (which we will denote o) of the frame, and a set of d linearly independent *basis vectors*.

In Fig. **??** we show a point p and vector $\overrightarrow{w}$. We have also given two coordinate frames, F and G. Observe that p and $\overrightarrow{w}$ can be expressed as functions of F and G as follows:

$$\begin{aligned} p &= 3 \cdot F.\vec{e}_0 + 2 \cdot F.\vec{e}_1 + F.o \\ \overrightarrow{w} &= 2 \cdot F.\vec{e}_0 + 1 \cdot F.\vec{e}_1 \end{aligned}$$

$$\begin{aligned} p &= 1 \cdot G.\vec{e}_0 + 2 \cdot G.\vec{e}_1 + G.o \\ \overrightarrow{w} &= -1 \cdot G.\vec{e}_0 + 0 \cdot G.\vec{e}_1 \end{aligned}$$

Notice that the position of $\overrightarrow{w}$ is immaterial, because in affine geometry vectors are free to float where they like.



$p_{[F]} = (3, 2, 1)$ $w_{[F]} = (2, 1, 0)$ $p_{[G]} = (1, 2, 1)$ $w_{[F]} = (-1, 0, 0)$

Fig. 2: Coordinate Frames.

Coordinate Axiom and Homogeneous Coordinates: Recall that our goal was to represent both points and vectors as a list of scalar values. To put this on a more formal footing, we introduce the following axiom.

Coordinate Axiom: For every point p in affine space, $0 \cdot p = \vec{0}$, and $1 \cdot p = p$.

This is a violation of our rules for affine geometry, but it is allowed just to make the notation easier to understand. Using this notation, we can now write the point and vector of the figure in the following way.

$$\begin{aligned} p &= 3 \cdot F.\vec{e}_0 + 2 \cdot F.\vec{e}_1 + 1 \cdot F.o \\ \overrightarrow{w} &= 2 \cdot F.\vec{e}_0 + 1 \cdot F.\vec{e}_1 + 0 \cdot F.o \end{aligned}$$

Thus, relative to the coordinate frame $F = \langle F.\vec{e}_0, F.\vec{e}_1, F.o \rangle$, we can express p and $\overrightarrow{w}$ as coordinate vectors relative to frame F as

$$p_{[F]} = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix} \quad \text{and} \quad \overrightarrow{w}_{[F]} = \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix}.$$

We will call these *homogeneous coordinates* relative to frame F. In some linear algebra conventions, vectors are written as row vectors and some as column vectors. We will stick with OpenGL's conventions, of using column vectors, but we may be sloppy from time to time.

As we said before, the term "vector" has two meanings: one as a *free vector* in an affine space, and now as a *coordinate vector*. Usually, it will be clear from context which meaning is intended.

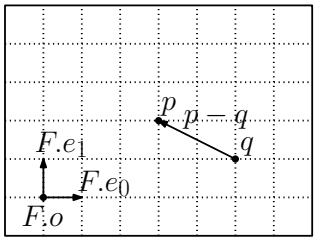
In general, to represent points and vectors in d-space, we will use coordinate vectors of length $d + 1$. Points have a last coordinate of 1, and vectors have a last coordinate of 0. Some authors put the homogenizing coordinate first rather than last. There are actually good reasons for doing this. But we will stick with standard engineering conventions and place it last.

Properties of homogeneous coordinates: The choice of appending a 1 for points and a 0 for vectors may seem to be a rather arbitrary choice. Why not just reverse them or use some other scalar values? The reason is that this particular choice has a number of nice properties with respect to geometric operations.

For example, consider two points p and q whose coordinate representations relative to some frame F are $p_{[F]} = (3, 2, 1)^\mathsf{T}$ and $q_{[F]} = (5, 1, 1)^\mathsf{T}$, respectively. Consider the vector

$$\vec{v} = p - q.$$

If we apply the difference rule that we defined last time for points, and then convert this vector into it coordinates relative to frame F, we find that $\vec{v}_{[F]} = (-2, 1, 0)^\mathsf{T}$. Thus, to compute the coordinates of $p - q$ we simply take the component-wise difference of the coordinate vectors for p and q. The 1-components nicely cancel out, to give a vector result (see Fig. **??**).



$$p_{[F]} = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix} \qquad q_{[F]} = \begin{pmatrix} 5 \\ 1 \\ 1 \end{pmatrix}$$

$$(p - q)_{[F]} = \begin{pmatrix} -2 \\ 1 \\ 0 \end{pmatrix}$$

Fig. 3: Coordinate arithmetic.

In general, a nice feature of this representation is the last coordinate behaves exactly as it should. Let u and v be either points or vectors. After a number of operations of the forms $u + v$ or $u - v$ or αu (when applied to the coordinates) we have:

- If the last coordinate is 1, then the result is a *point*.
- If the last coordinate is 0, then the result is a *vector*.
- Otherwise, this is not a legal affine operation.

This fact can be proved rigorously, but we won't worry about doing so.

This suggests how one might do type checking for a coordinate-free geometry system. Points and vectors are stored using a common base type, which simply consists of a 4-element array of scalars. We allow the programmer to perform any combination of standard vector operations on coordinates.

Just prior to assignment, check that the last coordinate is either 0 or 1, appropriate to the type of variable into which you are storing the result. This allows much more flexibility in creating expressions, such as:

$$\textit{centroid} \leftarrow \frac{p+q+r}{3},$$

which would otherwise fail type checking. (Unfortunately, this places the burden of checking on the run-time system. One approach is to define the run-time system so that type checking can be turned on and off. Leave it on when debugging and turn it off for the final version.)

Cross Product: The cross product is an important vector operation in 3-space. You are given two vectors and you want to find a third vector that is orthogonal to these two. This is handy in constructing coordinate frames with orthogonal bases. There is a nice operator in 3-space, which does this for us, called the *cross product*.

The cross product is usually defined in standard linear 3-space (since it applies to vectors, not points). So we will ignore the homogeneous coordinate here. Given two vectors in 3-space, $\vec{u}$ and $\vec{v}$, their *cross product* is defined as follows (see Fig. **??**(a)):

$$\vec{u} \times \vec{v} = \begin{pmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{pmatrix}.$$

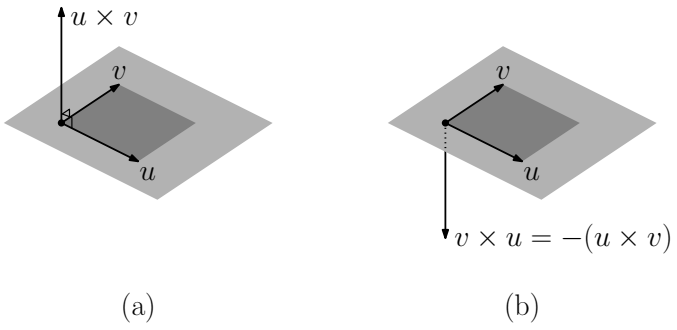


Fig. 4: Cross product.

A nice mnemonic device for remembering this formula, is to express it in terms of the following symbolic determinant:

$$\vec{u} \times \vec{v} = \begin{vmatrix} \vec{e}_x & \vec{e}_y & \vec{e}_z \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix}.$$

Here $\vec{e}_x$, $\vec{e}_y$, and $\vec{e}_z$ are the three coordinate unit vectors for the standard basis. Note that the cross product is only defined for a pair of free vectors and only in 3-space. Furthermore, we ignore the homogeneous coordinate here. The cross product has the following important properties:

Skew symmetric: $\vec{u} \times \vec{v} = -(\vec{v} \times \vec{u})$ (see Fig. **??**(b)). It follows immediately that $\vec{u} \times \vec{u} = 0$ (since it is equal to its own negation).

Nonassociative: Unlike most other products that arise in algebra, the cross product is *not* associative. That is

$$(\vec{u} \times \vec{v}) \times \vec{w} \neq \vec{u} \times (\vec{v} \times \vec{w}).$$

Bilinear: The cross product is linear in both arguments. For example:

$$\begin{aligned} \vec{u} \times (\alpha\vec{v}) &= \alpha(\vec{u} \times \vec{v}), \\ \vec{u} \times (\vec{v} + \vec{w}) &= (\vec{u} \times \vec{v}) + (\vec{u} \times \vec{w}). \end{aligned}$$

Perpendicular: If $\vec{u}$ and $\vec{v}$ are not linearly dependent, then $\vec{u} \times \vec{v}$ is perpendicular to $\vec{u}$ and $\vec{v}$, and is directed according the right-hand rule.

Angle and Area: The length of the cross product vector is related to the lengths of and angle between the vectors. In particular:

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

where θ is the angle between $\vec{u}$ and $\vec{v}$. The cross product is usually not used for computing angles because the dot product can be used to compute the cosine of the angle (in any dimension) and it can be computed more efficiently. This length is also equal to the area of the parallelogram whose sides are given by $\vec{u}$ and $\vec{v}$. This is often useful.

The cross product is commonly used in computer graphics for generating coordinate frames. Given two basis vectors for a frame, it is useful to generate a third vector that is orthogonal to the first two. The cross product does exactly this. It is also useful for generating surface normals. Given two tangent vectors for a surface, the cross product generate a vector that is normal to the surface.

Orientation: Given two real numbers p and q, there are three possible ways they may be ordered: $p < q$, $p = q$, or $p > q$. We may define an orientation function, which takes on the values $+1$, 0, or -1 in each of these cases. That is, $\text{Or}_1(p, q) = \text{sign}(q - p)$, where $\text{sign}(x)$ is either -1, 0, or $+1$ depending on whether x is negative, zero, or positive, respectively. An interesting question is whether it is possible to extend the notion of order to higher dimensions.

The answer is yes, but rather than comparing two points, in general we can define the orientation of $d + 1$ points in d-space. We define the *orientation* to be the sign of the determinant consisting of their homogeneous coordinates (with the homogenizing coordinate given first). For example, in the plane and 3-space the orientation of three points p, q, r is defined to be

$$\text{Or}_2(p, q, r) = \text{sign det} \begin{pmatrix} 1 & 1 & 1 \\ p_x & q_x & r_x \\ p_y & q_y & r_y \end{pmatrix}, \qquad \text{Or}_3(p, q, r, s) = \text{sign det} \begin{pmatrix} 1 & 1 & 1 & 1 \\ p_x & q_x & r_x & s_x \\ p_y & q_y & r_y & s_y \\ p_z & q_z & r_z & s_z \end{pmatrix}.$$

What does orientation mean intuitively? The orientation of three points in the plane is $+1$ if the triangle PQR is oriented counter-clockwise, -1 if clockwise, and 0 if all three points are collinear (see Fig. **??**). In 3-space, a positive orientation means that the points follow a right-handed screw, if you visit the points in the order $PQRS$. A negative orientation means a left-handed screw and zero orientation means that the points are coplanar. Note that the order of the arguments is significant. The orientation of (p, q, r) is the negation of the orientation of (p, r, q). As with determinants, the swap of any two elements reverses the sign of the orientation.

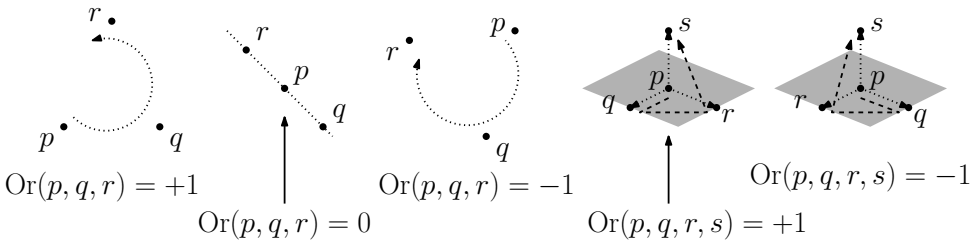


Fig. 5: Orientations in 2 and 3 dimensions.

You might ask why put the homogeneous coordinate first? The answer a mathematician would give you is that is really where it should be in the first place. If you put it last, then positive oriented things are "right-handed" in even dimensions and "left-handed" in odd dimensions. By putting it first, positively oriented things are always right-handed in orientation, which is more elegant. Putting the homogeneous coordinate last seems to be a convention that arose in engineering, and was adopted later by graphics people.

The value of the determinant itself is the area of the parallelogram defined by the vectors $q - p$ and $r - p$, and thus this determinant is also handy for computing areas and volumes. Later we will discuss other methods.