CMSC 425: Lecture 4
Geometry and Geometric Programming

Geometry for Game Programming and Graphics: For the next few lectures, we will discuss some of the basic elements of geometry. There are many areas of computer science that involve computation with geometric entities. This includes not only computer graphics, but also areas like computer-aided design, robotics, computer vision, and geographic information systems. While software systems like Unity provide support to do geometry for you, there are good reasons for learning this material. First, for those of you who will go on to design the successor to Unity, it is important to understand the fundamentals underlying gometric programming. Second, as a game programmer, you will find that there things that Unity *cannot* help with. In such cases, you will need to write scripts to do the geometry yourself.

Computer graphics deals largely with the geometry of lines and linear objects in 3-space, because light travels in straight lines. For example, here are some typical geometric problems that arise in designing programs for computer graphics.

Transformations: You are asked to render a twirling boomerang flying through the air. How would you represent the boomerang's rotation and translation over time in 3-dimensional space? How would you compute its exact position at a particular time?

Geometric Intersections: Given the same boomerang, how would you determine whether it has hit another object?

Orientation: You have been asked to design the AI for a non-player agent in a flight combat simulator. You detect the presence of a enemy aircraft in a certain direction. How should you rotate your aircraft to either attack (or escape from) this threat?

Change of coordinates: We know the position of an object on a table with respect to a coordinate system associated with the table. We know the position of the table with respect to a coordinate system associated with the room. What is the position of the object with respect to the coordinate system associated with the room?

Reflection and refraction: We would like to simulate the way that light reflects off of shiny objects and refracts through transparent objects.

Geometric Interpolationn: We want to smoothly interpolate between two different positions and/or two different rotations in space.

Such basic geometric problems are fundamental to computer graphics, and over the next few lectures, our goal will be to present the tools needed to answer these sorts of questions. There are various formal geometric systems that arise naturally in game programming and computer graphics. The principal ones are:

Affine Geometry: The geometry of simple "flat things": points, lines, planes, line segments, triangles, etc. There is no defined notion of distance, angles, or orientations, however.

Euclidean Geometry: The geometric system that is most familiar to us. It enhances affine geometry by adding notions such as distances, angles, and orientations (such as clockwise and counterclockwise).

Projective Geometry: In Euclidean geometry, there is no notion of infinity (in the same way that in standard arithmetic, you cannot divide by zero). But in graphics, we often need to deal with infinity. (For example, two parallel lines in 3-dimensional space can meet at a common *vanishing point* in a perspective rendering. Think of the point in the distance where two perfectly straight train tracks appear to meet. Computing this vanishing point involves points at infinity.) Projective geometry permits this.

Affine Geometry: Affine geometry is basic to all geometric processing. It's basic elements are:

- *Scalars*, which we can just think of as being real numbers
- *Points*, which define locations in space
- *Free vectors* (or simply *vectors*), which are used to specify direction and magnitude, but have no fixed position.

The term "free" means that vectors do not necessarily emanate from some position (like the origin), but float freely about in space. There is a special vector called the *zero vector*, $\vec{0}$, that has no magnitude, such that $\vec{v} + \vec{0} = \vec{0} + \vec{v} = \vec{v}$.

Note that we did *not* define a *zero point* or "origin" for affine space. This is an intentional omission. No point special compared to any other point. (We will eventually have to break down and define an origin in order to have a coordinate system for our points, but this is a purely representational necessity, not an intrinsic feature of affine space.)

You might ask, why make a distinction between points and vectors?[1] Although both can be represented in the same way as a list of coordinates, they represent very different concepts. For example, points would be appropriate for representing a vertex of a mesh, the center of mass of an object, the point of contact between two colliding objects. In contrast, a vector would be appropriate for representing the velocity of a moving object, the vector normal to a surface, the axis about which a rotating object is spinning. (As computer scientists the idea of different abstract objects sharing a common representation should be familiar. For example, stacks and queues are two different abstract data types, but they can both be represented as a 1-dimensional array.)

Because points and vectors are conceptually different, it is not surprising that the operations that can be applied to them are different. For example, it makes perfect sense to multiply a vector and a scalar. Geometrically, this corresponds to stretching the vector by this amount. It also makes sense to add two vectors together. This involves the usual head-to-tail rule, which you learn in linear algebra. It is not so clear, however, what it means to multiply a point by a scalar. (For example, the top of the Washington monument is a point. What would it mean to multiply this point by 2?) On the other hand, it does make sense to add a vector to a point. For example, if a vector points straight up and is three meters long, then adding this to the top of the Washington monument would naturally give you a point that is three meters above the top of the monument.

We will use the following notational conventions. Points will usually be denoted by lower-case Roman letters such as p, q, and r. Vectors will usually be denoted with lower-case Roman letters, such as u, v, and w, and often to emphasize this we will add an arrow (e.g., $\vec{u}$, $\vec{v}$, $\vec{w}$).

[1] Unity does not distinguish between them. The data type Vector3 is used to represent both points and vectors.

Scalars will be represented as lower case Greek letters (e.g., α, β, γ). In our programs, scalars will be translated to Roman (e.g., a, b, c). (We will sometimes violate these conventions, however. For example, we may use c to denote the center point of a circle or r to denote the scalar radius of a circle.)

Affine Operations: The table below lists the valid combinations of scalars, points, and vectors. The formal definitions are pretty much what you would expect. Vector operations are applied in the same way that you learned in linear algebra. For example, vectors are added in the usual "tail-to-head" manner (see Fig. 1). The difference $p - q$ of two points results in a free vector directed from q to p. Point-vector addition $r + \vec{v}$ is defined to be the translation of r by displacement $\vec{v}$. Note that some operations (e.g. scalar-point multiplication, and addition of points) are explicitly not defined.

$vector \leftarrow scalar \cdot vector$,	$vector \leftarrow vector/scalar$	scalar-vector multiplication
$vector \leftarrow vector + vector$,	$vector \leftarrow vector - vector$	vector-vector addition
$vector \leftarrow point - point$		point-point difference
$point \leftarrow point + vector$,	$point \leftarrow point - vector$	point-vector addition

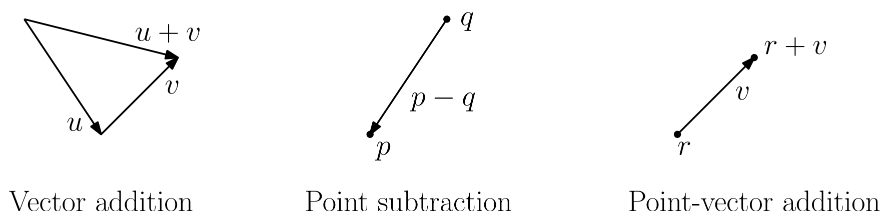

Fig. 1: Affine operations.

Affine Combinations: Although the algebra of affine geometry has been careful to disallow point addition and scalar multiplication of points, there is a particular combination of two points that we will consider legal. The operation is called an *affine combination*.

Let's say that we have two points p and q and want to compute their midpoint r, or more generally a point r that subdivides the line segment $\overline{pq}$ into the proportions α and $1 - \alpha$, for some $\alpha \in [0, 1]$. (The case $\alpha = 1/2$ is the case of the midpoint). This could be done by taking the vector $q - p$, scaling it by α, and then adding the result to p. That is,

$$r = p + \alpha(q - p),$$

(see Fig. 2(a)). Another way to think of this point r is as a *weighted average* of the endpoints p and q. Thinking of r in these terms, we might be tempted to rewrite the above formula in the following (technically illegal) manner:

$$r = (1 - \alpha)p + \alpha q,$$

(see Fig. 2(b)). Observe that as α ranges from 0 to 1, the point r ranges along the line segment from p to q. In fact, we may allow to become negative in which case r lies to the

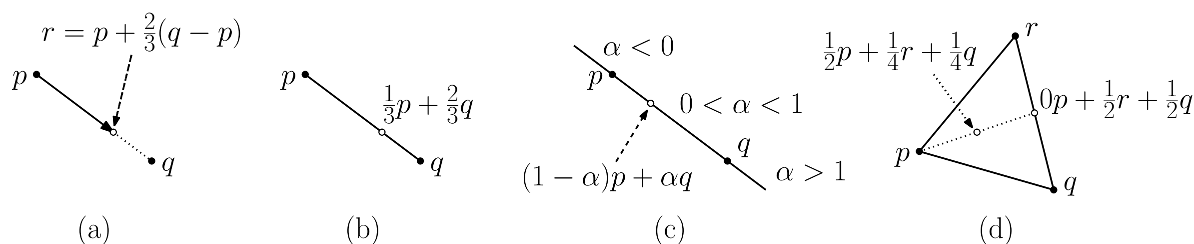

Fig. 2: Affine combinations.

left of p, and if $\alpha > 1$, then r lies to the right of q (see Fig. 2(c)). The special case when $0 \leq \alpha \leq 1$, this is called a *convex combination*.

In general, we define the following two operations for points in affine space.

Affine combination: Given a sequence of points $p_1, p_2, \ldots, p_n$, an affine combination is any sum of the form

$$\alpha_1 p_1 + \alpha_2 p_2 + \ldots + \alpha_n p_n,$$

where $\alpha_1, \alpha_2, \ldots, \alpha_n$ are scalars satisfying $\sum_i \alpha_i = 1$.

Convex combination: Is an affine combination, where in addition we have $\alpha_i \geq 0$ for $1 \leq i \leq n$.

Affine and convex combinations have a number of nice uses in graphics. For example, any three noncollinear points determine a plane. There is a 1–1 correspondence between the points on this plane and the affine combinations of these three points. Similarly, there is a 1–1 correspondence between the points in the triangle determined by the these points and the convex combinations of the points (see Fig. 2(d)). In particular, the point $\frac{1}{3}p + \frac{1}{3}q + \frac{1}{3}r$ is the *centroid* of the triangle.

We will sometimes be sloppy, and write expressions like $\frac{1}{2}(p+q)$, which really means $\frac{1}{2}p + \frac{1}{2}q$. We will allow this sort of abuse of notation provided that it is clear that there is a legal affine combination that underlies this operation.

To see whether you understand the notation, consider the following questions. Given three points in the 3-space, what is the union of all their affine combinations? (Ans: the plane containing the 3 points.) What is the union of all their convex combinations? (Ans: The triangle defined by the three points and its interior.)

Euclidean Geometry: In affine geometry we have provided no way to talk about angles or distances. Euclidean geometry is an extension of affine geometry which includes one additional operation, called the *inner product*.

The inner product is an operator that maps two vectors to a scalar. The product of $\vec{u}$ and $\vec{v}$ is denoted commonly denoted $(\vec{u}, \vec{v})$. There are many ways of defining the inner product, but any legal definition should satisfy the following requirements

Positiveness: $(\vec{u}, \vec{u}) \geq 0$ and $(\vec{u}, \vec{u}) = 0$ if and only if $\vec{u} = \vec{0}$.

Symmetry: $(\vec{u}, \vec{v}) = (\vec{v}, \vec{u})$.

Bilinearity: $(\vec{u}, \vec{v} + \vec{w}) = (\vec{u}, \vec{v}) + (\vec{u}, \vec{w})$, and $(\vec{u}, \alpha\vec{v}) = \alpha(\vec{u}, \vec{v})$. (Notice that the symmetric forms follow by symmetry.)

See a book on linear algebra for more information. We will focus on a the most familiar inner product, called the *dot product*. To define this, we will need to get our hands dirty with coordinates. Suppose that the d-dimensional vector $\vec{u}$ is represented by the coordinate vector $(u_0, u_1, \ldots, u_{d-1})$. Then define

$$\vec{u} \cdot \vec{v} = \sum_{i=0}^{d-1} u_i v_i,$$

Note that inner (and hence dot) product is defined only for vectors, not for points.

Using the dot product we may define a number of concepts, which are not defined in regular affine geometry (see Fig. 3). Note that these concepts generalize to all dimensions.

Length: of a vector $\vec{v}$ is defined to be $\sqrt{\vec{v} \cdot \vec{v}}$, and is denoted by $\|\vec{v}\|$ (or as $|\vec{v}|$).

Normalization: Given any nonzero vector $\vec{v}$, define the *normalization* to be a vector of unit length that points in the same direction as $\vec{v}$, that is, $\vec{v}/\|\vec{v}\|$. We will denote this by $\widehat{v}$.

Distance between points: $\text{dist}(p, q) = \|p - q\|$.

Angle: between two nonzero vectors $\vec{u}$ and $\vec{v}$ (ranging from 0 to π) is

$$\text{ang}(\vec{u}, \vec{v}) \;=\; \cos^{-1}\left(\frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\|\|\vec{v}\|}\right) \;=\; \cos^{-1}(\widehat{u} \cdot \widehat{v}).$$

This is easy to derive from the law of cosines. Note that this does not provide us with a signed angle. We cannot tell whether $\vec{u}$ is clockwise our counterclockwise relative to $\vec{v}$. We will discuss signed angles when we consider the cross-product.

Orthogonality: $\vec{u}$ and $\vec{v}$ are *orthogonal* (or perpendicular) if $\vec{u} \cdot \vec{v} = 0$.

Orthogonal projection: Given a vector $\vec{u}$ and a nonzero vector $\vec{v}$, it is often convenient to decompose $\vec{u}$ into the sum of two vectors $\vec{u} = \vec{u}_1 + \vec{u}_2$, such that $\vec{u}_1$ is parallel to $\vec{v}$ and $\vec{u}_2$ is orthogonal to $\vec{v}$.

$$\vec{u}_1 \;\leftarrow\; \frac{(\vec{u} \cdot \vec{v})}{(\vec{v} \cdot \vec{v})}\vec{v}, \qquad\qquad \vec{u}_2 \;\leftarrow\; \vec{u} - \vec{u}_1.$$

(As an exercise, verify that $\vec{u}_2$ is orthogonal to $\vec{v}$.) Note that we can ignore the denominator if we know that $\vec{v}$ is already normalized to unit length. The vector $\vec{u}_1$ is called the *orthogonal projection* of $\vec{u}$ onto $\vec{v}$. If we think of $\vec{v}$ as being a normal vector to a plane, then the projection of u onto this plane is called *orthogonal complement* of u with respect to v, and is given by $\vec{u}_2 \leftarrow \vec{u} - \vec{u}_1$.

Doing it with Unity: Unity does not distinguish between points and vectors. Both are represented using Vector3. Unity supports many of the vector operations by overloading operators. Given vectors u, v, and w, all of type Vector3, the following operators are supported:

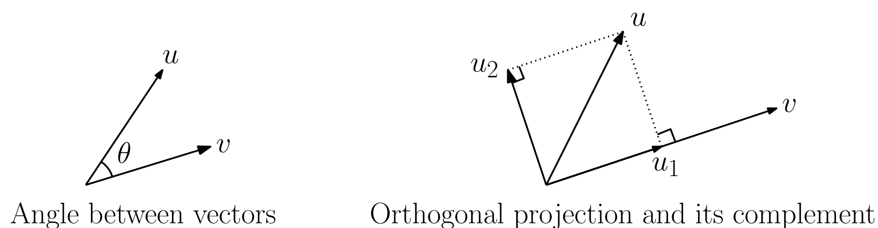

Fig. 3: The dot product and its uses.

```
u = v + w; // vector addition
u = v - w; // vector subtraction
if (u == v || u != w) { ... } // vector comparison
u = v * 2.0f; // scalar multiplication
v = w / 2.0f; // scalar division
```

You can access the components of a Vector3 using as either using axis names, such as, u.x, u.y, and u.z, or through indexing, such as u[0], u[1], and u[2].

The Vector3 class also has the following members and static functions.

```
float x = v.magnitude; // length of v
Vector3 u = v.normalize; // unit vector in v's direction
float a = Vector3.Angle (u, v); // angle (degrees) between u and v
float b = Vector3.Dot (u, v); // dot product between u and v
Vector3 u1 = Vector3.Project (u, v); // orthog proj of u onto v
Vector3 u2 = Vector3.ProjectOnPlane (u, v); // orthogonal complement
```

Some of the Vector3 functions apply when the objects are interpreted as points. Let p and q be points declared to be of type Vector3. The function Vector3.Lerp is short for *linear interpolation*. It is essentially a two-point special case of a convex combination. (The combination parameter is assumed to lie between 0 and 1.)

```
float b = Vector3.Distance (p, q); // distance between p and q
Vector3 midpoint = Vector3.Lerp(p, q, 0.5f); // convex combination
```