CS 161 Week 2

Aidan Jan

April 19, 2023

Summary of Scalar, Point, and Vector Operations

Summary of Scalar, Point & Vector Ops

Red font = makes sense for affine, does not make sense for linear operations

Operands	Operands	Add (+)	Subtract (-)	Multiply (*)
Scalar-Scalar	s_1, s_2	$s = s_1 + s_2$	$s = s_1 - s_2$	$s = s_1 * s_2$
Point-Point	P_1, P_2	$P = a_1*P_1 + a_2*P_2$	$v = P_2 - P_1$	X
Vector-Vector	v_1, v_2	$v = v_1 + v_2$	$v = v_1 - v_2$	X
Scalar-Point	s, P_1	X	X	$P = s * P_1$
Scalar-Vector	s, v_1	X	X	$v = s * v_1$
Point-Vector	P_1, v_1	$P_2 = P_1 + v_1$	$P_2 = P_1 - v_1$	X

Affine and Convex Combinations

Lines can be parametrically defined based on points, in the format:

$$p = \alpha_1 \cdot p_1 + \alpha_2 \cdot p_2 + ... + \alpha_n \cdot p_n$$

This does not make much sense when each point can be anywhere, and each α can be of any value. Typically, we would define

$$\alpha_1 + \alpha_2 + ... + \alpha_n = 1$$

In this case, the point generated would be an interpolation between the points $p_1, p_2, ..., p_n$.

For example: consider the case with only two points, p_1, p_2. If we define a new point, $p_3 = \alpha_1 \cdot p_1 + \alpha_2 \cdot p_2$, and $\alpha_1 + \alpha_2 = 1$, then setting some alpha value would guarantee that p_3 lies on the line between p_1 and p_2. In fact, the α_1 value determines how far along the line p_3 lies. If $\alpha_1 = 0.5$, then that implies that $\alpha_2 = 0.5$. In this case, p_3 would lie on the midpoint between p_1 and p_2.

The constraint that the α values sum to 1 is called the **Affine Constraint**, and this form of finding a point location is a **Affine Combination of Points**.

$$\sum_{i=0}^{n} \alpha_i = 1$$

The **Convex Constraint** is the constraint that states that all α values must be greater than 0 and sum to 1. Without this constraint, p_3 may lie on the line passing through p_1 and p_2, however, it is not guaranteed to be between the two points. If α_1 is negative, it would be in the opposite direction from p_1 to p_2, and if α_2 were negative, the point would be on the line, but beyond p_2.

$$\alpha_i > 0 \, \forall \, i \in [0, n]$$

Combinations of points that meet both of these constraints are known as **Convex Combinations of Points**.

A similar operations can be done with higher-dimensional points, and also with an infinite number of points.

Vectors

Vectors are denoted with angle brackets $\langle\rangle$, with n number of arguments, where n is the number of dimensions the vector is in. Vectors have an x-component and a y-component, in other words, a direction and a magnitude. However, it does not have a beginning location. Thus, it is often assumed vectors are based at the origin. In this case, a vector can be defined by a point. For example, the point (7, 4) would define the vector $\langle 7, 4 \rangle$, which represents moving 7 units in the x-direction and 4 units in the y-direction.

With vectors, we can define **basis**, in colloquial terms, a different coordinate system. The "default" Cartesian coordinate plane is defined by vectors $x = \langle 1, 0 \rangle$ and $y = \langle 0, 1 \rangle$, since to move one unit in the x-direction means to multiply the x-direction vector by one, and the y-direction vector by zero.

If a plane were defined by $x = \langle 8, 6 \rangle$ and $y = \langle 5, 10 \rangle$, then to move 1 unit in the x-direction would be the equivalent of moving 8 units in x and 6 units in y, relative to the "default" Cartesian coordinate plane. Thus, (1, 0) in our newly created coordinate plane would be (8, 6) in the Cartesian coordinate plane. The two vectors used to define a plane need not be perpendicular, although it is preferable to have them perpendicular.

To define spaces in more than two dimensions, we would use as n vectors to define the space, where n is the number of dimensions. Also, each vector used in the definition would include n arguments.

A matrix multiplication can be done to change basis, and the operation for this is called a **change of basis.**

Points versus Vectors

Vectors do not have a location, only the two components that tell where to move. In this case, how do we tell if

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

is a vector or a point? In this case, we use what is called **homogenizing vector**, which features an extra number, called a **homogenizing scalar** (rather than using a Cartesian vector). It is the bottom number in a matrix. If it is a one, then it is a point. If it is a zero, then it is a vector. For example, if

$$\begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}$$

was given, then we would know that it is a vector, while

$$\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$$

would be a point.

Do Homogeneous Vectors Work with Operations?

The short answer: Yes. If we subtract two points, we expect to find a vector.

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} - \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 - x_2 \\ y_1 - y_2 \\ 0 \end{bmatrix}$$

This is correct; it does produce a vector.
If we try adding two vectors, we expect to get a vector:

$$\begin{bmatrix} x_1 \\ y_1 \\ 0 \end{bmatrix} + \begin{bmatrix} x_2 \\ y_2 \\ 0 \end{bmatrix} = \begin{bmatrix} x_1 + x_2 \\ y_1 + y_2 \\ 0 \end{bmatrix}$$

All other operations also work! This is also consistent with the fact it's impossible to add two points or scale a point. Thus, the homogeneous scalar is always 0 or 1.

Vectors in Graphics

One of the common ways to use vectors in graphics is to create gradients. Suppose you have two points where you want to have an uniform gradient from red to blue in. You can define a vector parametrically. Let p_1 have the color red (#FF0000), and p_2 have the color blue (#0000FF). Of course, p_1 and p_2 also have their locations.
We can define a parametric equation as the following:

$$p = p_1 \cdot (t) + p_2 \cdot (1 - t),$$

where t is a parameter between 0 and 1. As we move the point from p_1 to p_2, we also multiply the color by the parameter. For each point between p_1 and p_2, we would have a smooth gradient of color.

If instead we wish to color an area rather than just a line, we simply use more points. If we want a triangle with gradients of red, green, and blue, we may define $p_1 = (0, 0, \#FF0000)$, $p_2 = (500, 0, \#00FF00)$, and $p_3 = (250, 250\sqrt{3}, \#0000FF)$. For a given point inside the triangle, we can define the equation

$$p = p_1 \cdot t + p_2 \cdot u + p_3 \cdot (1 - u - v),$$

where u and v are parameters.

A similar approach can be done with more points or more dimensions.

Dot Products of Vectors

To compute a dot product between two vectors, sum the products of the vector components.

$$\langle a, b\rangle \cdot \langle c, d\rangle = ac + bd$$

Notice that the result of a dot product is a **scalar**, not a vector.
The dot product of two **perpendicular** vectors is always 0.
The formal definition is

$$w \cdot v = \sum_{i=1}^{n} w_i \cdot v_i,$$

where w and v are both vectors with n components.

Some other properties of Dot Product are as follows:

1. Symmetry: $a \cdot b = b \cdot a$
2. Linearity: $(a + b) \cdot c = a \cdot c + b \cdot c$
3. Homogeneity: $(sa) \cdot b = s(a \cdot b)$
4. $|b|^2 = b \cdot b$
5. $a \cdot b = |a| \cdot |b| \cdot \cos(\theta)$

Additionally, if $a \cdot b > 0$, then the angle between a and b is acute. If greater than 0, then the angle is obtuse.

Cross Products of Vectors

Cross product is calculated by cross-multiplying the vector components. It is only defined for 3D vectors. To cross product two vectors, $a = \langle x_1, y_1, z_1\rangle$, $b = \langle x_2, y_2, z_2\rangle$,

$$a \times b = (y_1 \cdot z_2 - z_1 \cdot y_2)i + (z_1 \cdot x_2 - x_1 \cdot z_2)j + (x_1 \cdot b_2 - x_2 \cdot b_1)k,$$

where i, j, k are the vector components. The cross product can also be written in matrix form:

$$a \times b = \begin{vmatrix} i & j & k \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}$$

Modeling

Through a process called **tessellation** we can convert any 3-D surface into a large number of polygons. (Usually triangles). The smaller the polygons the higher quality the end product.

There are multiple types of polygons:

1. Closed/open
2. Wireframe/filled
3. Planar/non-planar
4. Convex/concave
5. Simple/non-simple

Open polygons do not form a closed shape, wireframe polygons only include the edges, filled polygons are filled, planar polygons can lie within a 2-D plane, Convex polygons always have internal angles below 180 degrees on the vertices, convex polygons may not. Finally, non-simple polygons may pass through itself.

Triangles are the most often used because they are closed, planar, convex, and simple.

3D Shapes

To store 3D shapes, we can also represent them using polygons. For example, consider a cube. A vertex list can be created

1. (0, 1, 1)
2. (1, 1, 1)
3. (0, 0, 1)
4. (1, 0, 1)
5. (0, 1, 0)
6. (1, 1, 0)
7. (0, 0, 0)
8. (1, 0, 0)

From here, we can create a list of faces:

1. 0, 2, 3, 1
2. 1, 3, 7, 5
3. 5, 7, 6, 4
4. 4, 6, 2, 0

5. 4, 0, 1, 5

6. 2, 6, 7, 3

Notice that when representing the faces, all the vertices are read in counter-clockwise order, from the outside of the object. This is important because more than one polygon may share the same vertices. However, the downside for this is that once a 3-D shape is tessellated, it loses its inside; it becomes a surface.