Vector Mathematics Basics

Although Unity stores x, y & z coordinates in a Vector3 construct, it can represent both **points** and **vectors**. A point is a location in space. A vector is a direction and length. A vector has no predetermined starting point. There is only one point in space that has its coordinates, however vectors with the same values can be anywhere. The x,y,z of the point is a single location in space, the x,y,z of a vector is the length of the vector in each of those dimensions.

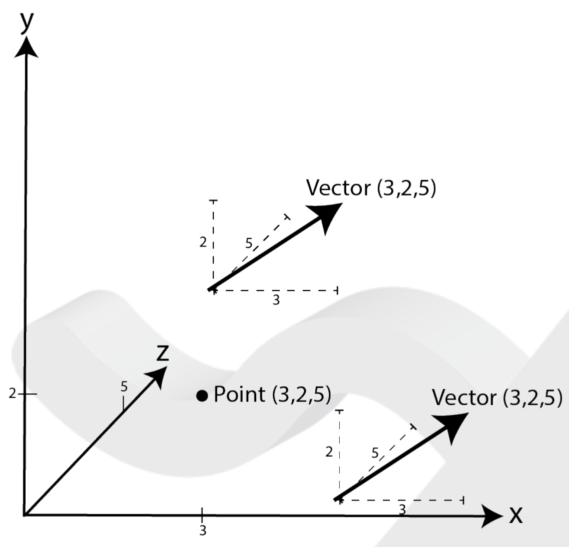

Fig1. Vectors Versus Points

1. Adding and Subtracting Vectors and Points

When adding vectors each of the x, y and z coordinates are added together in turn. E.g. (2,3,5) + (1,3,5) will equal (3,6,10). It's the same if adding a point and a vector. The point (2,4,5) + the vector (3,2,2) will result in the point (5,6,7).

a **point** plus a **vector** will result in a point
a **vector** plus a **vector** will result in a vector

In games we add vectors to points to move objects around. For example, the movement of characters from one location to another. The vector calculated between one point and another tells us the direction and distance.

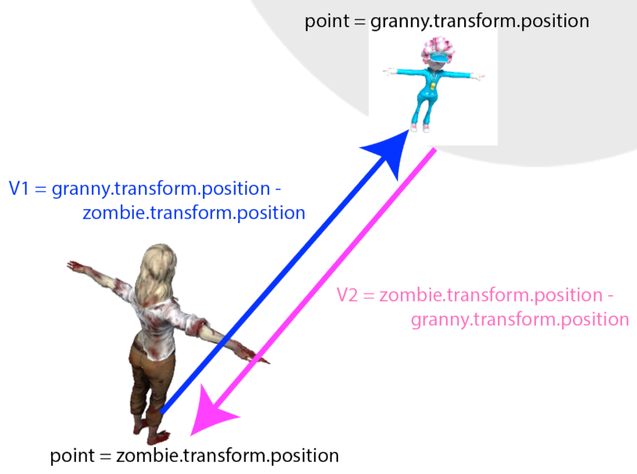
point = granny.transform.position
V1 = granny.transform.position - zombie.transform.position
V2 = zombie.transform.position - granny.transform.position
point = zombie.transform.position

Fig 2. Calculating Vectors Between Points

In *Fig 2*. If Stevie (the zombie) wants to get from her position to Granny's position then the vector she must travel is granny's position minus Stevie's position. For example, if Stevie was at (10,15, 5) and Granny was at (20, 15, 20) then the vector from Stevie to Granny would be (20, 15, 20) - (10, 15, 5) = (10, 0, 15). If Granny wanted to find her vector to Stevie the equation is reversed.

Then to move Stevie from her current location to that of Granny you would add the vector (V1) to Stevie's position thus: (10,15, 5) + (10, 0, 15) = (20, 15, 20) which you can check is correct because it's the position of Granny!

Vectors can also be added together to give a total direction and magnitude. I always think of this like the old treasure map instructions of take 5 steps North and 3 steps East. In this case you could take the vector to the North and then the vector to the East, or you could travel as the crow flies and just take the vector that is the result of adding both together as shown by the red vector in *Fig 3*.

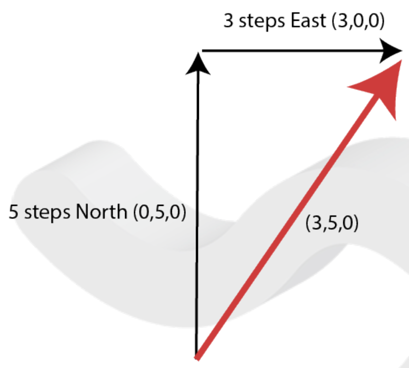

Fig 3. Adding Vectors

In Unity, you can add and subtract vectors in a mathematical equation thus:

```
Vector3 v1 = new Vector3(5,6,7);
Vector3 v2 = new Vector3(1,2,3);
Vector3 v3 = v1 + v2;
```

or

```
stevie.transform.position = new Vector3(10,15,5);
granny.transform.position = new Vector3(20,15,20);
Vector3 directionToGranny = granny.transform.position -
                            stevie.transform.position;
stevie.transform.position += directionToGranny;
```

2. Scaling Vectors

Scaling a vector increases or decreases its length but *does not* change its direction. We might change the length of a vector in a game to calculate a new position that is in the same direction but further away.

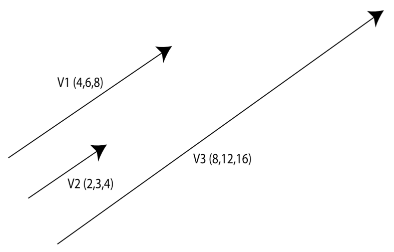

Fig 4. Scaling Vectors

In *Fig 4*. V1 has been scaled by half to get V2 which is half V1's length whereas V3 represents V1 doubled. Scaling a vector is a matter of multiplying or dividing it by a value. If you multiply it by a value less than a whole number it will have the effect of making it smaller. For example,

V2 = V1 * 0.5;

You can also do this operation like this:

V2 = V1 / 2;

In Unity scaling vectors is also achieved through straight forward mathematics such as:

```
Vector3 v1 = new Vector3(5,6,7);
Vector3 v2 = v1 * 0.5f;
```

or

```
Vector3 v1 = new Vector3(5,6,7);
float scalingFactor = 2.0f;
Vector3 v2 = v1 * scalingFactor;
```

3. Opposite Vectors

When a vector is multiplied or divided by a negative number achieves the effect of turning the vector around to point in the opposite direction. If you multiply a vector by -1 it will turn the vector around but leave the length exactly the same as shown in *Fig 5*. V2 is the exact opposite of V1 and is achieved by multiplying V1 * -1.

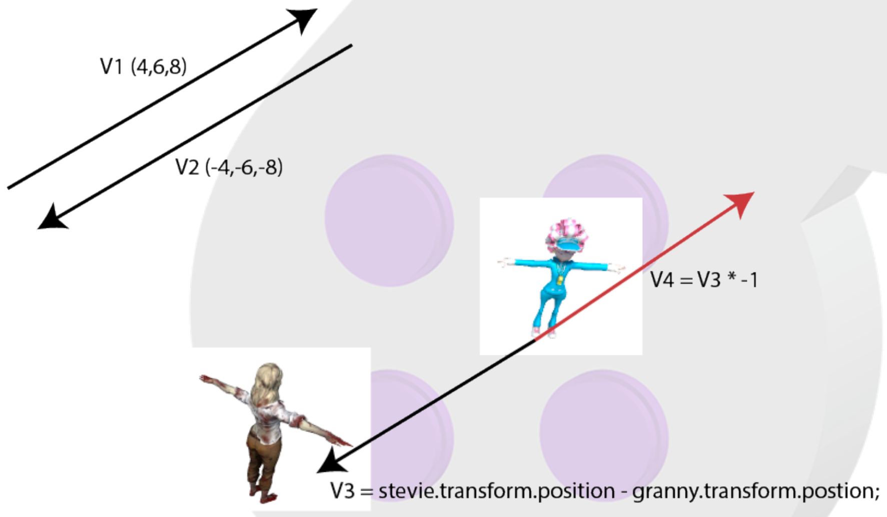

Fig 5. Opposite Vectors

Opposite vectors are useful for calculating characters escape routes. For example, if Granny wanted to get away from Stevie, then if she worked out the vector toward Stevie and multiplied it by -1 she'd be moving away instead of toward.

4. Distances

The length of a vector is called its magnitude. When the direction toward a character is calculated as we've done in the previous examples, by taking one position away from another, the resulting length of that vector is the distance between the characters.

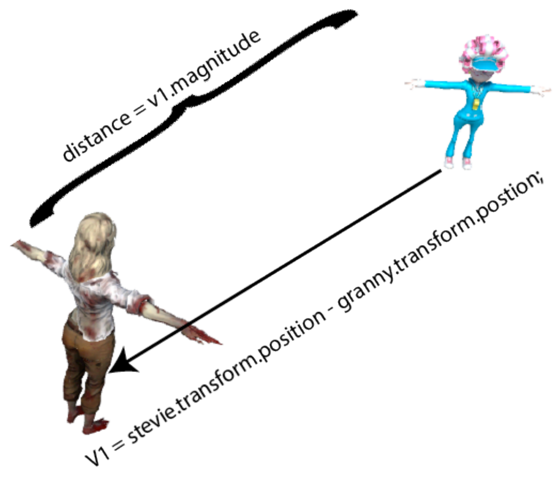

Fig 6. Calculating Distances Between Points

In games distance between locations is used by decision making AI as well as moving objects around. For example, an NPC might work out the distance to a player before deciding whether to attack or not.

In Unity the length of a vector is found by accessing its magnitude property thus:

```
Vector3 v1 = new Vector3(20,35,56);
float length = v1.magnitude;
```

or

```
stevie.transform.position = new Vector3(10,15,5);
granny.transform.position = new Vector3(20,15,20);
Vector3 directionToGranny = granny.transform.position -
                            stevie.transform.position;
float distanceBetweenStevieGranny = directionToGranny.magnitude;
```

You can also calculate the distance between two points using the *Vector3.Distance()* method thus:

```
float distanceBetweenStevieGranny = Vector3.Distance(stevie.transform.position,
                                                     granny.transform.position);
```

5. Angles

In determining the direction in which to travel to get from one location to another you might also require an angle that indicates how much a character needs to turn to be facing that location, otherwise you'll get a character that moves sideways.

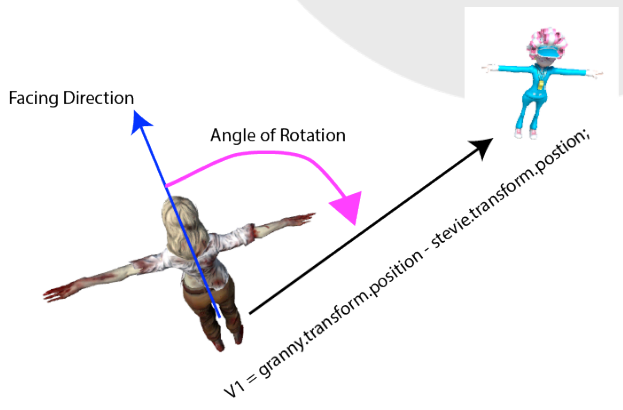

Fig 7. Angle of Rotation

Once you have calculated the angle between the way the character is facing and the direction it is about to travel you'll be able to program its turning.

Without the use of an angle Unity can snap a character around to face in a particular direction with its *LookAt()* method. For example:

```
stevie.transform.LookAt(granny.transform.position);
```

Though this is not desirable if you want to be able to see the character turning little by little. To do that you need to apply a *Slerp()* thus:

```
Vector3 direction = granny.transform.position - stevie.transform.position;
stevie.transform.rotation = Quaternion.Slerp(stevie.transform.rotation,
                            Quaternion.LookRotation(direction),
                            Time.deltaTime * rotationSpeed);
```

This code will turn Stevie through the angle of rotation little by little (if this code were placed in an update) at the rate of the *rotationSpeed* until she is facing Granny. You'll see this *a lot* throughout this course.

In Unity, if you want to get your hands on an actual angle you can use the *Angle()* method thus:

```
Vector3 direction = granny.transform.position - stevie.transform.position;
float angleOfRotation = Vector3.Angle(stevie.transform.forward, direction);
```

In games you find the angle not only useful for working out how a character should turn but also in preparing for future character behaviour. Some characters might be programmed not to see the player if they can't see outside 30 degrees of their forward vector while others, like NPC cars, might use the upcoming angle in the corner of a road to determine how much to apply the brakes.