Chapter 3

Direction & Length

Scaling a vector is a pretty neat trick. We can add half an arrow, or double an arrow, or use 0-1 to slide along the length of an arrow. But that trick can't give us distances.

If we want to travel 5 units along an arrow, or move along it at 2 units/second, we need more math.

The first trick is getting the length of an arrow. The second is getting a length 1 version of an arrow. After a bit, we'll start thinking of any arrow as really being those two parts – the direction, and how far.

3.1 Magnitude/distance

The way to find the distance between two things is to make an arrow between them, then measure the length. All distances are really measuring how long an arrow is, which is officially called its *magnitude*.

To find the distance between us and the green cube we make the offset arrow, as usual, then measure it:

```
Vector3 toGreen=greenCube.position-transform.position;
float dist = toGreen.magnitude;
```

Distances and offsets can feel similar. An offset gives us the distance over x,y,z. But not the straight line real distance. Where-as distance is a single number – the actual distance – but doesn't tell us how to get there.

Since magnitude simply measures arrows, we can check it by making an arrow out of numbers, then measuring it:

```
Vector3 A = new Vector3(3,0,4);
float dist = A.magnitude; // 5
```

You may remember that answer from school. 3 over and 4 up is a triangle with the arrow as the hypotenuse. It's length 5 by the Pythagorean theorem. That's the math the system is doing for us.

Here are some fun facts about distance and magnitude:

- Distance is always one positive number. Negative distance makes no sense. If it's 3 miles from my house to the quarry, it's 3 miles from the quarry to my house, not negative 3. Offsets can be negative – (3,0,0) to the quarry and (-3,0,0) back.
- The subtraction order doesn't matter for computing distance. `A-B` is a backwards arrow from `B-A`, but they're each the same length.
- Logically, `magnitude` is for offsets, not positions. `(transform.position).magnitude` isn't an error, but it thinks of our position as an arrow from (0,0,0). It says how far you are from (000), which isn't a helpful number.
- **magnitude** is the official mathematical term for the length of an arrow. Not just for computers, but for real math.
- Not having parens after `A.magnitude` is a (required) shortcut. It's really a function call running an equation.

A fun distance problem is getting a "flat" distance. We often want to know how far two things are apart on a map, not counting hills:

```
                   _B_
                  /
                / hill
__A_____________    o
  |- we want this --|
```

The trick is to create the arrow we want to measure. We can find the B-A arrow, then change the y part of it to 0:

```
Vector3 arrow = B-A;
arrow.y=0; // now it's from A to the dot
float dist = arrow.magnitude;
```

Since distance is never negative, `if(dist<3)` is way to check if you're close enough. If something were 10 units behind you, the distance would be a positive 10 and the "close enough" check would fail.

Here's a basic "find nearest enemy from a list" test using a simple distance compare:

```
float shortestDist=99999;
int nearest=-1;
```

```
for(int i=0; i<Things.length; i++) {
  float dist=(Things[i]-transform.position).magnitude;
  if(dist<shortestDist) { // <- simple less-than distance check
    nearest=i;
    shortestDist=dist;
  }
}
```

3.1.1 Direction arrows

It's often good enough to have an arrow pointing at the target. We don't care how long it is, only the direction it aims. In fact, if we have an arrow pointing from A to B, we often prefer it to be length 1, no matter how far apart they are. To get to B, we'll go that direction until we get there.

`transform.forward` is a great example of a direction arrow. It's an offset, but not really. Offsets tell us how to find one specific spot. transform.forward is a way to walk forward as much as we need, in our forward Direction.

We can think of a direction vector as half of an offset. Like an offset, it has to start from a position, but it doesn't go to any particular spot yet. We have to also know the distance.

Some functions say they take a Direction as an input. They're talking about this sort of direction. They want a Vector3 which is correctly aimed, but just any length. For a direction input, (2,1,0) and (4,2,0) are the same. But (2,-1,0) or (1,2,0) aren't – they aim in a completely different direction.

Raycasts and Direction

The built-in raycast function takes a direction as an input, giving a nice example of the concept. A Raycast is a standard 3D game trick. In the game world you draw an imaginary line, as far as you want, and check whether it hits any cubes or cows. It's used to see what your laser hits, or check if you can move in a certain direction. Technically, a line going in only 1 direction, not both, is called a ray, thus the name raycast.

Here's a raycast that checks for obstacles 5 units in front of us:

```
Vector3 lookDir = transform.forward;
if(Physics.Raycast(transform.position, lookDir, 5)
  print("forward 5 is blocked");
```

Hopefully the 3 inputs are obvious: starts from us, in our forward direction, for 5 units. `transform.forward` is just the direction. It's length is 1, but that doesn't matter since it's a direction. The distance is all by itself in the next variable.

Suppose we want to look for obstacles mostly ahead, but with a small tilt right. A cheap way of getting a tilt is to think of it as a slope – 1 right for every 12 forward.

To make it more interesting, we'll to look in that direction for 7 steps, then, if it's clear, for 20 steps:

```
// the length doesn't matter, only the angle:
Vector3 lookDir = transform.forward*12 + transform.right;

if(Physics.Raycast(transform.position, lookDir, 7) {
  if(Physics.Raycast(transform.position, lookDir, 20) {
    print("lots of room");
  }
  else print("some room");
}
else print("not enough room -- need at least 7");
```

I think that looks pretty nice. We use the same direction each time, with distances of 7 and then 20. Seeing the distance to all by itself makes it more readable. And letting us make the direction however we want make that first line short and clear.

There are other ways. Sometimes the second input is the ending point. Sometimes it's a real offset going the total distance to check. But the direction + distance method is also common.

3.1.2 Normalized direction

The most useful direction vectors have exactly length 1. We've seen this with `transform.forward`. To walk 1.8 steps forward, we can use `transform.forward*1.8f`. The math term is *normalized direction*, or sometimes *unit vector* (shorthand for "a 1-unit long vector").

This is so common that when something asks for for a direction, most people check whether it needs to be length 1. Often they make it length 1 just in case. Or, to be really safe, convert all of your directions to length 1 as soon as you get them.

There's a really slick trick to take any arrow and rescale it to be length 1. All we have to do is divide by its length. We can get the length using `magnitude`, so the whole thing is too easy. This gets a length 1 arrow from us to the green cube:

```
// this part is a repeat:
Vector3 toGreen = greenCube.position - transform.position;
float dist = toGreen.magnitude;

toGreen = toGreen/dist; // toGreen is now length 1
```

Now `transform.position + toGreen` is exactly 1 away from us, towards green.

The trick works for any arrow. If the arrow is longer than 1, it shrinks. If it was shorter, it grows. It works for backwards aimed arrows, or going along only x. The only thing it won't work for is (000), since that's not a direction.

The way it works is so clever. We know any arrow times 2 is the same arrow, twice as long. An arrow divided by 2 is aimed the same way, but 1/2 as long. So, a length 4.71 arrow divided by 4.71 is going the same direction with length 4.71/4.71 = 1!

The official math term for "make a length 1 version of an arrow" is *normalizing*. If you've seen Normals used to affect lighting in 3D models, this is completely different. There's no relation at all – it's only a coincidence both terms use the same word.

Unity provides a function to normalize. All it does is compute the length and divides by it. We can do that ourselves, especially if we already needed to know the length. But doing it the official way can look nice.

In comes in the 2 standard versions: change yourself, or don't and return the result:

```
A = new Vector3(3,4,0); // sample vector

B = A.normalized; // B is (0.6, 0.8, 0), A is unchanged
A.Normalize(); // A is now (0.6, 0.8, 0)
```

We don't really need the second one, since `A=A.normalized;` does the same thing, but some people like how it looks.

Some normalizing notes:

- (1,0,0) is normalized. It's length 1.
- (0,-5,0) normalizes to (0,-1,0). The negative has to stay, or else it wouldn't be the same direction. (1,0,1) normalizes to (0.707, 0, 0.707). That's length one because of trigonometry.
- If you accidentally normalize something a few extra times, no problem. Normalizing a length 1 vector does nothing. You're dividing by 1.
- If you already know the length, `A=A/len;` is faster than `A.Normalize()`. But many people don't know (or do know but don't remember) the division trick. Writing the official commands can make code easier to read.
- The one thing you can't normalize is (0,0,0), since that's no direction. There's no possible length 1 version of that, since it's not pointing anywhere. The Unity normalize command gives you back (0,0,0). That's not correct, but it's the best it can do.

For real you often get a direction by taking B-A when they're at the same spot. There's no direction to go to B, since you don't need one – you're there.

3.2 Normalized direction + length

When we used `transform.position + transform.forward*1.5f` we were taking advantage of how transform.forward is length one. We could get a position exactly 1.5 units ahead of us. We couldn't do that with regular offsets before. Now we can, by making them into unit vectors.

Suppose we want to walk 2.5 meters towards the green cube:

```
Vector3 uToGreen = (greenCube.position - transform.position).normalized;

Vector3 pos = transform.position + uToGreen*2.5f;
```

The 2nd line looks exactly like the math we did with `transform.forward` and friends, since it is. We're scaling it by 2.5, but since it's length 1, that makes it exactly 2.5 long, towards green.

The first line is just our two old lines combined. (B-A) is the arrow, and normalized makes it be length 1. The u in `uToGreen` is for unit. Everyone knows unit means 1 unit long.

Previously we were able to put a cube behind us, hiding from green, but the best we could do for distance was 10% of the whole line. Now we can put it exactly 3 meters behind us:

```
redCube.position = transform.position - uToGreen*3;
```

Again, it should look a lot like our forward/right/up math, since it is. Starting from us, we're going 3 meters away from green.

3.2.1 Using both

The full trick is splitting the offset into 2 parts: the unit vector, and distance. Together, those are better than just the full offset. Our old code doing this, all in one place:

```
// getting ready to solve any hard offset problem with uToGreen and dist:
Vector3 toGreen = greenCube.position - transform.position;
float dist = toGreen.magnitude;
Vector3 uToGreen = toGreen.normalized;
```

Fun fact: `uToGreen*dist` is the original arrow.

With these two variables we can place the red cube 2 units past the green one (it will appear to be hiding from us, behind green):

```
// us ---------------- green -- red

redCube.position = transform.position + uToGreen*(dist+2);
```

Pretty slick, right? From us, we walk `dist` units along the arrow, which puts us on green, then 2 more. I almost hate to ruin it by saying that an easier way is to start at the green cube and walk 2 more: `greenCube.position + uToGreen*2`.

Or we can leave it between us, but almost to green – like green is using it as a shield. We can either walk `dist-3` units towards green, or start at green and walk backwards 3 units:

```
// us ------------------ red --- green

redCube.position = transform.position + uToGreen*(dist-3);

// or shorter version, start at red, walk 3 back to us:
redCube.position = greenCube.position - uToGreen*3;
```

With our new unit vectors and lengths we can make the red cube travel from us to Green, better than it did before. Now we can make it start 1 unit ahead of us, stop when it gets within 1, and move at a consistent speed:

```
void Update() {
  // pretend we have uToGreen and len

  dist+=0.05f; if(dist>len-1) dist=1; // goes from 1 to len-1
  redCube.position = transform.position + uToGreen*dist;
}
```

The first line controls the actual distance away from us. That makes it super easy to start and stop exactly as far away as we want to. The motion is now at a constant speed, which means it finally takes longer to go further.

The second line is basic *unitVector*distance* positioning. Figuring out the correct values for `dist` was all the work.

3.2.2 More throwing with rigidbodies

Using unit vectors to toss rigidbodies is just as easy. We did it before, in the spaceship, using `transform.forward`. Now we can do it with `uToGreen`.

I'll pretend the red cube has a rigidbody. The space key will put it 2 units ahead us of, then launch it towards the green cube:

```
void Update() {
if(Input.GetKeyDown(KeyCode.Space)) {
  // find the spot 2 units in my forward:
```

```
  Vector3 rPos = transform.position + transform.forward*2;

  redCube.position = rPos;

  // motion is long line from red to green, at speed 10:
  Vector3 toGreen = greenCube.position - rPos;
  Vector3 uToGreen = toGreen.normalized;
  redCube.GetComponent<Rigidbody>().velocity = uToGreen*10;
}
```

Once you know that velocity is set once, then moves us automatically, and it's in units/second, and that `uToGreen` is a unit vector, then `velocity = uToGreen*10` should be simple enough.

3.3 Looking at the numbers

If you look at the numbers for distance, they can seem funny. One way to solve this is not to look at them. But if you can't help yourself, here are some notes:

When you have a long length and a short one, the short one counts for almost nothing. For example (10,1,0), has a length of only 10.05.
To see how that makes sense, imagine the triangle: 10 over and 1 up. The diagonal is clearly just a little more than 10 long.

Even when the numbers are close together, the answer is smaller than it seems. (5,4,0) has a length of 6.4. In 3D, the numbers are even shorter. The arrow (3,4,5) has a length of only 7.1. That's the longest, 5, plus not much more for the diagonals.

If you estimate distance between two points in your head, you can think of all differences as positive. For example, comparing (10,10,10) to (2,13,9). All that matters is: 8 away, 3 away and 1 away. The distance will be 8 plus a little more.

Normalized arrows have the same funny-looking math as `transform.forward`. Any 45-degree unit vector looks like (0.71, 0.71, 0). If you normalize (1,1,0), that's what you get. A unit arrow at 30 degrees really is (0.6, 0, 0.8). Almost all unit arrows are wrong-looking numbers like that.

The actual equation for distance is the 3D pythagorean theorem:
$x^2 + y^2 + z^2 = D^2$. In computer code:

```
float dist = Mathf.Sqrt(x*x + y*y + z*z);
```