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

As a shortcut, Unity lets us measure distance either way: length of an arrow, or distance between two points. `Vector3.Distance(A,B)` or `C.magnitude` (no parens, just because.) Examples:

```
float dist = Vector3.Distance(transform.position, marker.position);
print("You are "+dist+" away from the marker");

// same thing: get arrow to marker, then measure it:
Vector3 toMarker=marker.position-transform.position; // arrow from us to marker
dist = toMarker.magnitude; // length of arrow = distance
```

The 1-line call to `Distance` looks nicer since it's a shortcut. Inside, it's subtracting the points and running `magnitude`.

We don't have to use these to measure between objects. They work on any points or arrows, even ones we just make. You might remember this from the pythagorean theorem (a right triangle with sides 3 and 4 has hypotenuse 5):

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

`A`'s not really an arrow – we didn't subtract 2 points to get it. Maybe we intend to use it as one .... But either way `magnitude` gives the length as if it were an arrow.

Here are some fun facts about distance and magnitude:

- Distance is always one positive number. Negative distance make 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 order in `Distance` doesn't matter. `Vector3.Distance(A,B)` is the same as `Vector3.Distance(B,A);`
- `magnitude` is for offsets. If you use it with a point, like `(transform.position).magnitude`, it gives the distance from (0,0,0), which is rarely useful. Plus, a clearer way is `Vector3.Distance(transform.position, Vector3.zero)`.
- You can get flat xz distance (distance on a map, not counting hills) by changing the arrow's y to 0:

  ```
  Vector3 toMarker = marker.position-transform.position;
  toMarker.y=0; // now toMarker is a flat arrow
  float dist = toMarker.magnitude;
  ```

- Just so you know, **magnitude** is the official mathematical term for the length of an arrow.
- Also just so you know, not having ()-parens after `A.magnitude` is the official rule. It's a real function call, but using `C#`'s getter trick to leave them out.

3.1.1 Direction

Often we don't need an arrow going all the way to the target. We need one pointing to the target, which we often call a direction arrow. To simplify, direction arrows are usually length 1. For example, `transform.forward` is our forward direction arrow.

The important thing is that direction arrows only tell us which way to go. The length is unimportant. For example (2,1,0) and (4,2,0) are the same direction. If you wanted to write that direction (twice as far x as y) the official way,

you should use trig to make it length 1, but don't have to.

Raycasts are a nice example of using direction arrows. They take a position and a direction and walk that way until they hit something. This shoots an imaginary ray north from us:

```
Vector3 dir=new Vector3(0,0,10);
if(Physics.Raycast(transform.position, dir))
  print("forward is blocked");
```

Raycasts think the second input is a direction. (0,0,10) is the forward, +z direction. It would say we were blocked if anything was 1 in front of us, or 10 or any distance. We could have used (0,0,1) for the direction and it would run the same.

If we wanted to check for obstacles sideways and up, `dir` could be (2,1,0) or (4,2,0).

For a comparison, `Debug.DrawRay` takes an actual offset. It starts at the point you give it, then adds the offset and draws exactly that arrow. In this case, the 10 really means 10:

```
Vector3 dir=new Vector3(0,0,10);
Debug.DrawRay(transform.position, dir); // draws length 10 forward arrow
dir=new Vector3(0,0,0.333f);
Debug.DrawRay(transform.position, dir); // draws very short forward arrow
```

Having Raycast and DrawRay work differently is for sure confusing. But it does a nice job of showing the terms: offset means we care about the whole arrow and where the tip ends, and direction means we don't.

(Funny story: in the manual DrawRay says it takes a direction, but that's a typo. It's an offset.)

3.1.2 Normalized direction

You can often use a direction arrow of any length, but there are some tricks you can do with an arrow of exactly length 1. The math term for that is a normalized direction, or sometimes a unit vector (which is shorthand for "a 1-unit long vector.")

There's a really slick trick to turn an arrow into length 1: divide by its length. Here's an example getting a length 1 direction to a marker:

```
Vector3 toMarker = marker.position - transform.position;
float dist = toMarker.magnitude;
Vector dirToMarker = toMarker/dist; // length 1 arrow to marker
```

This trick works for any arrow – it can be pointing backwards, or have length less than 1 (it will grow) – and it still works.

Unity provides two shortcut functions for that. One of them makes *you* be length one, and another makes a length one version of you. Examples:

```
A = new Vector3(3,4,0);
A.Normalize(); // A is now (0.6, 0.8, 0), which happens to be length 1

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

A=A.normalized; // same as A.Normalize();
```

Normalizing is the ten dollar math term for getting a length 1 version of an arrow. But it's nothing special besides dividing by the length. For example, (1,0,0) is normalized, which is a fancy way of saying it's already length 1.

Some normalizing notes:

- Normalize something that's already length 1 doesn't change it. It doesn't do any harm, either – it's safe to normalize something just in case.
- Most people use the official `normalize` command. But if you already know the length, `A=A/len;` works fine and is faster.
- 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. Unity just gives you (0,0,0), which isn't correct, but it's the best it can do.

3.2 Normalized direction + length

With the theory out of the way, we're ready to do tricks by breaking an offset into length one direction and magnitude. We start with this:

```
Vector3 toB = B-A;
float len = toB.magnitude;
toB = toB.normalized;
```

Now `toB` is a length 1 direction arrow towards B, and `len` is how far.

The simplest trick is that `A+toB` is *one* unit from A towards B. `A+toB*3.5f` is exactly 3.5 units from A towards B. We can pick the exact distance in `A+toB*dist`. We can slide `dist` from 0 to the total, `len`, to walk a real distance from A to B.

Here are few simple examples. This puts a "shield" two units away from us, facing the marker:

```
// get length 1 arrow to marker, all in one line:
Vector3 toMarker = (marker.position-transform.position).normalized;
```

```
shield.position=transform.position+toMarker*2;
shield.LookAt(marker); // not needed, but fun
```

You might remember from before we could use a fraction of a vector to do something similar. The improvement here is we can give an actual distance. Before the best we could do was 1/10th of an arrow, which could grow and shrink.

An earlier example had a cube hiding behind us. The same math works here:

```
Vector3 toMarker = (marker.position-transform.position).normalized;
// hiding from marker, behind us and 3 away:
hidingCube.position=transform.position - toMarker*3;
```

This next example slides a ball from the marker to us. We did that before, but now it moves at a constant rate (before, it took 2 seconds, no matter how close or far we were):

```
public Transform ball;
float ballDist=0; // this is the actual distance from us, in units

void Update() {
  Vector3 toMarker = marker.position-transform.position;
  float len=toMarker.magnitude;
  toMarker.Normalize();

  ball.position=transform.position+toMarker*ballDist;

  // slide ballDist from 0 to total len, at 2 units/sec:
  ballDist+=2*Time.deltaTime;
  if(ballDist>len) ballDist=0;

}
```

If you remember, the old way used the entire arrow, and the variable was a 0 to 1 percent. This way, the variable is the real distance we want to be. Neither way is better, but this one often looks nicer. If we're 8.5 away from the marker, we have to wait while the distance slowly goes from 0 to 8.5.

The unit vector trick is also great for setting velocity. If you remember, I previously used `velocity=transform.forward*5` to fire a ball. That was really using the unit vector trick – relying on how `transform.forward` is always length 1.

If you want to shoot in a direction towards something you can do it by manually computing the unit vector. This has the space key shoot a ball towards a marker. It starts 2 in front of us and flies at 5 units/second:

```
if(Input.GetKeyDown(" ")) { // space key fires
  // standard spawn. Assume ballPrefab is set up:
  Transform ball = Instantiate(ballPrefab);
  ball.position = transform.position + transform.forward*2

  Vector3 toMarker=(marker.position-ball.position).normalized;

  Vector3 vel = toMarker*5; // <- key line. unit vector times speed
  ball.GetComponent<Rigidbody>().velocity = vel;
}
```

Notice how if finds the total arrow from the *ball* to the marker, not from the player. Otherwise the angle might be off. And we don't bother computing the distance to the marker, since we didn't need it.

If you're turned away from the target this will shoot the ball back through you. And, same as before, it works better if gravity is turned off on the ball prefab.

3.3 Looking at the numbers

Sometimes you look at the numbers for distance, and they seem funny. If you never do, skip this.

A surprising thing is, 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. Going up by one added just 0.05 to the distance. If you flip it around, this makes sense: imagine driving to a town 10 miles east and 1 mile north. That's 11 miles if we have to drive that way. But we know a diagonal straight-shot road will be a good deal – it will be just a little longer than 10 miles.

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. The answer had to be at least 5, and the 3 and 4 didn't add much.

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. So it's like an arrow (8,3,1). The distance will be 8 plus a little more.

For real, distances are computed using the Pythagorean theorem: $x^2 + y^2 = d^2$. That's why (3,4,0) has length 5.

Normalized (length one) vectors have the same funny-looking math as `transform.forward`. A unit diagonal arrow really is (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.

But, to repeat myself, you don't need to know these numbers. If you know trig or want to learn, it's fun to look at them. If you notice (0.6, 0.8, 0) and think "wait, aren't they suppose to be length 1?" now you know they don't add to one – they pythagorean square add to 1.