Chapter 6

Moving

We already know the math to use points and offsets to gradually move things. Unity has some standard built-in commands that do the same thing. Since they make it so easy, we can try some fancier movement.

We know how to move simple rotations – spinning around y. Unity has some standard quaternion commands that let us gradually move rotations in more ways.

6.1 Speed per second

If we want nice movement we need a better way of saying how fast we go. "Add 0.05 each update, and we're going whatever speed that works out to" is bad. We should set our speeds using amount-per-second.

The math seems easy enough – there are 60 update/second, so we divide by 60. But sometimes there are 30/second. Even worse, that could change mid-program, or even mid second.

The simplest way to pro-rate movement is using the actual amount of time an update takes. We don't know how long this one will take, so we use the time since the previous.

At the start of each update, Unity computes how long it's been since the last one started, and puts it in `Time.deltaTime`. That's a good name, since delta means the amount of change.

This code adds 3.5 to x every second:

```
public float x;

void Update() {
  x += 3.5f * Time.deltaTime;
}
```

It's pretty slick. If things run at a perfect 60fps, this is the same as dividing by 60. But even if updates have random times between them, this always adds the proportion of 3.5 based on how much time passed. Each second's worth of updates always adds to 3.5.

Whenever we want anything to change over time, we'll write down the amount per second, then multiply by `Time.deltaTime` when we use it.

An example, this moves us forward at 3.5 units/second:

```
void Update() {
  transform.position += transform.forward*3.5f*Time.deltaTime;
}
```

In our minds, `transform.forward*3.5f` is the speed each second. `Time.deltaTime` computes the tiny fraction to add this update.

This lets us enter any speed/second, and moves the red cube towards us at that speed. It uses our old friend, the normalization trick:

```
public float speed; // per second
public Transform redCube;

void Update() {
  Vector3 toUs = transform.position - redCube.position;
  toUs = toUs.normalized;
  toUs = toUs*speed; // our movement arrow in 1 second

  redCube.position += toUs*Time.deltaTime;
}
```

To use this trick properly, just make sure your speed is per second. Suppose you have `pct+=0.01f;`. That seems like just a guess at how much per frame. We'll throw that out. We really want it to go by 0.5/second.

The nicer math is `pct+=0.5f*Time.deltaTime;`.

Likewise, suppose you see `n1*Time.deltaTime` just anywhere in code. You know `n1` is something per second.

6.2 MoveTowards

Unity (and lots of other people) has a helpful function that takes a starting point, an ending point and a distance. It tells you where you'd be if you went that far along the line.

As a bonus, it won't overshoot. If A and B are 3 apart, and you tell it to move 4, it stops at B.

We can write it using stuff we already know:

```
Vector3 MoveTowards(Vector3 A, Vector3 B, float dist) {
  Vector3 wholeArrow=B-A;

  // if we would overshoot, tell them we stop on B:
  if(wholeArrow.magnitude<dist) return B;

  // standard normalization to go a distance:
  Vector3 unitArrow = wholeArrow.normalized;
  return A+unitArrow*dist;
}
```

It's called MoveTowards, but clearly doesn't move anything. Even so, it's pretty useful when we want to move something towards a point.

Here's a redo of moving the red cube to us, using the real built-in MoveTowards:

```
redBall.position = Vector3.MoveTowards(
    redBall.position, transform.position, speed*Time.deltaTime);
```

Notice how we finally have `Time.deltaTime`. MoveTowards takes the actual distance to move. Our `speed` is in per-seconds, and we need to convert it to the movement for this one step.

This use is the most common. We take our current position as the start. Next frame we're in a new spot, which is the start of the next MoveTowards.

This also lets us track a moving target. If we move around, or even teleport, the red cube starts moving towards there.

The movement can seem robotic, but only because we use the same speed all the time. A fun thing to do is change it. This makes us go faster when we're further away (and slower as we get close):

```
  // speed is 1/2 the distance to the target:
  speed = ((goHere.position - transform.position).magnitude)/2;
  // but at least 1:
  if(speed<1) speed=1;

  // same line as before:
  redCube.position=Vector3.MoveTowards(
    redCube.position, transform.position, speed*Time.deltaTime);
```

We could do the opposite and have it start at speed 0, but quickly increase (somewhere else would reset speed to 0 for each fresh target):

```
  speed += 3.0f*Time.deltaTime;
```

```
// same line as before:
redCube.position=Vector3.MoveTowards(
  redCube.position, transform.position, speed*Time.deltaTime);
```

Notice the extra use of `Time.deltaTime` in the first line. We want to increase our speed by 3 each second. `Time.deltaTime` lets us add it gradually each frame.

A summary. MoveTowards is good when:

- The place you want to go is a specific point.

  We often just want to move in a direction. That's easy and we don't need MoveTowards for it.

- You want to move in units/second.

  The other way to move is percent-based: you'll get there in 2 seconds, at whatever speed that takes. This way takes whatever time it takes, based on the speed you use.

- You don't want to overshoot.

  Sometimes the target is only an aiming guide, and you want to go through it and past. We can do that easier the old way: compute our movement once at the start, and add it each frame.

Basically, there are lots of ways to move. MoveTowards does a nice job handling one common way.

6.2.1 Lerp

Another common way to move between two points is by using a 0-1 percent. There's a common built-in function that can help:

```
Vector3 Lerp(Vector3 start, Vector3 end, float pct) {
  if(pct<0) pct=0; if(pct>1) pct=1;
  Vector3 arrow=end-start;
  return start+arrow*pct;
}
```

Forcing the percent to be between 0 and 1 is new, but otherwise it's merely basic offset math. But that's fine. A 2-line function with a nice name is a good deal. Lerp stands for *linear interpolation*, which is the math term for this.

A fun non-movement-based example, suppose we need cats to randomly appear on one edge of the screen. We could find the position like this:

```
Vector3 catPos = Vector3.Lerp(sideLow, sideHigh, Random.value);
// note: Random.value is a 0-1 shortcut
```

That's a nice, math-free way of saying "anywhere on a line between those two spots." It's easier to control the positioning using 0-1: fully random is easy to read, if we want a lower-half cat 0 to 0.5 is the obvious way, and so on.

Lerp for movement works the same as we did it earlier. We need to save the original starting point, and will move a percent from 0 to 1:

```
 saved
original    current
 start      position                target
   S ----------o-----------------------E
  0.0           ^ currently at         1.0
                 0.3 of the way
```

The hardest part is probably getting the speed working correctly. With percent movement we say how many seconds it takes. But then our math needs to flip it – if it takes 4 seconds, we need to go 0.25 each second.

A mini lerp-base move between two points:

```
void Update() {
  pct += (1.0f/seconds)*Time.deltaTime;
  redCube.position = Vector3.Lerp(start, end, pct);

  if(pct>=1.0f) // do whatever we do when we get there
}
```

As we already know, this moves faster or slower depending how far away. It also works funny if we move `end` midway through (our position suddenly shifts to be on the new line, the same percent of the way.)

Percent-based motion is nice for displays: a far-away coal mine has a faster moving arrow.

This also shows why Unity's standard Lerp limits pct to 0-1. Legally, a 1.1 lerp is 10% past the end, which is no problem. We've done it. But motion code don't usually want that. Unity's version lets us overshoot `pct` and stop exactly on the end.

There's another, completely different way people use Lerp. Sometimes you want a fun "shoot off and slow down" motion, like collecting a cherry and having it fly into your inventory.

The main think is we don't care about the speed or the time it takes. We only care about a fun motion. It works like this:

```
void Update() {
  cherry.position = Vector3.Lerp(cherry.position, endPoint, 0.05f);
```

Notice how the percent never changes, but the starting position does. It says to always move us 5% closer. At 60 times/frame, that adds up fast. It never gets to the end, but it gets within a rounding error pretty fast.

When you see a Lerp where the thing before the `=` matches the start, it's this fun version.

And, to repeat, this method is no good for exact numbers. If you need to go 10 units in 2 seconds, there's no math for that. The best you can do is play with the 0.05. And even if you throw in `Time.deltaTime`, this way runs slower on slower framerates.

It's just an easy way to get a special effect.

6.3 Gradually rotating

Our previous method of making a moving rotation had us change `degrees` and then re-compute the quaternion: `degrees+=2; q=Quaternion.Euler(0,degrees,0);`. It's nice to know we can use that if we have to. But there are lots of shortcuts.

There's also a new thing: we can take any two rotations and smoothly spin from one to the other.

6.3.1 Spinning an arrow

This command rotates an arrow. The new thing is we don't need to tell it how to rotate. Instead we give it the final arrow and it figures out the best way to spin to it. To visualize it, imagine both arrows coming from the same place.

The basic code looks like this:

```
arrow = Vector3.RotateTowards(arrow, endArrow, 0.02f, 0);
```

If we run this over and over, `arrow` will eventually turn into `endArrow`. The third input is the amount to spin. Sadly, it's in radians – 1 radian is about 60 degrees.

If `arrow` is 1 unit forward, and `endArrow` is 1 unit right, this does a simple 90 degree spin over y, which we could do before.

The cool part is the arrows can be in any crazy direction. It figures out the best way to spin. Imagine the arrows are two points on a globe – the spin will be the best way to fly between them.

Here's a longer real example. The 'A' key puts the ball 4 units in front of us, then it automatically spins to be 4 units straight up. If we turn ourselves in various interesting ways, we'll see it rotate along the nearest side:

```
public Transform ball;
Vector3 arrow;

void Update() {
  arrow=Vector3.RotateTowards(
```

```
      arrow, Vector3.up*4, 1.5f*Time.deltaTime, 0);

  ball.position = transform.position+arrow;

  // reset, for testing:
  if(Input.GetKeyDown(KeyCode.A)) arrow=transform.forward*4;
}
```

The speed of 1.5 is the radian issue. It translates to 90 degrees, which is the rotation per second.

A note: for arrows close together, this looks like MoveTowards. Put your fingers out in a V. MoveTowards goes straight from the tip of the first to the second. RotateTowards spins the finger. You end in the same place, but there's a little arc.

But aim two fingers nearly opposite. MoveTowards goes straight through the middle. RotateTowards make a big half-circle, going the shortest way.

If the ending arrow is a different length, it will also change the length as it spins. The 4th input is how fast.

A problem is they probably won't synch up. It might finish turning and grow the rest in place, or grow to full length quickly while only partly turned. If you need them to finish at the same time, you'll need to do the math yourself. But it usually looks fine if it's even close.

6.3.2 Moving a rotation

Rotations have a similar move-towards-like function. You give it the current rotation, the target and how much to move (in degrees, this time.) Here's a simple example of spinning us towards a target:

```
Quaternion qWant = Quaternion.Euler(-90,0,0); // straight up

void Update() {
  transform.rotation = Quaternion.RotateTowards(
      transform.rotation, qWant, 60*Time.deltaTime);
}
```

This spins us from our current facing to facing up, spinning at 60 degrees/second. Try it with us starting in various odd directions. Like the arrow `RotateTowards`, it rotates us the closest way.

But this also takes into account the z-roll. Roll into your back and run it – we'll spin straight up while also rolling right-side-up. For more fun, point yourself straight up but z-rolled. Running will spin you in place until you're straightened out.

The degrees per second counts the change in aiming and rolling. If you're not changing your facing much, but have to spin on z a lot, it will take longer than it seems.

Here's a practical example. We want to look at the red cube by smoothly spinning:

```
void Update() {
  Vector3 qWant=Quaternion.LookRotation(
      redCube.position-transform.position);

  transform.rotation=Quaternion.RotateTowards(
      transform.rotation, qWant, 30*Time.deltaTime);
}
```

This handles moving targets fine, since it always moves a little from our current facing. We'll rotate at a constant speed, but that won't look too bad.

Here's one more which I use sometimes. We have an object that normally only turns on y, but sometimes it gets knocked down. We want it to "get up":

```
float yFacing; // our normal y-spin

bool getUp() { // call this every frame until you get up
  // the standing up rotation:
  Vector3 qWant=Quaternion.Euler(0,yFacing,0);

  transform.rotation=Quaternion.RotateTowards(
      transform.rotation, qWant, 90*Time.deltaTime);

  // standing has our personal y as (0,1,0)
  // if y is almost 1, close enough:
  return transform.up.y>0.99f;
}
```

This isn't different than any other RotateTowards, but it looks pretty cool how we get back up to how we were facing.

6.3.3 Rotation lerp

Rotations have their own Lerp. Like the one we've seen for vectors, it takes the start rotation, the end, and a 0-1 percent.

The main use is to get in-between rotations or fractions of rotations. To get a rotation 1/2-way between `q1` and `q2`, use:

```
transform.rotation = Quaternion.Lerp(q1, q2, 0.5f);
// rotation halfway between
```

Suppose `q1` is a look-rotation to a cat, and `q2` aims you at a dog. The line above computes a rotation looking half-way between them.

If you need to cut an angle in half, a trick is to average it with no-rotation. This computes 1/2 of `q`:

```
Quaternion qHalf=Quaternion.Lerp(Quaternion.identity, q, 0.5f);
```

You might be wondering why we don't use `q/2` or `(q1+q2)*0.5f`. They aren't legal math, and even if they were, they wouldn't do rotation stuff. `Quaternion.Lerp` is the substitute.

Suppose you want to write `q2/4`. Use `Quaternion.Lerp(Quaternion.identity, q2, 0.25f);`. It's a a pain, but not too bad.

Like movement lerps, Lerp for quaternions can be used for takes-this-long spins. This spins you from start to end in 3 seconds:

```
  pct01+=(1/3.0f)*Time.deltaTime; // 0.333 each second
  transform.rotation=Quaternion.Lerp(startRotation, endRotation, pct0to1);
  if(pct01>=1) { // do done with rotation stuff
```

Quaternions can also use the Lerp fast-then-slow trick to make pretty spins. This gives a nice hurry-then-slowdown spin to make us look straight up. Notice how the percent is always 0.03, but it starts from our current spin:

```
Quaternion qUp; // pretend this was set to aiming up

void Update() {
    transform.rotation=
      Quaternion.Lerp(transform.rotation, qUp, 0.03f);
  }
}
```

This can be nice if you were going to snap to a rotation, and that would be fine. But you want something just a little nicer.

6.4 Misc, summary

Both Lerps (points and rotations) have an unclamped version, which means the percent can be 1.1 or -0.5. For points you hardly need it, since `p1-p2*0.5f+p2*1.1f;` is easier to write.

But for rotations the unclamped version is helpful. For example we can scale `q` by 1.5 or -0.1:

```
q=Quaternion.LerpUnclamped(Quaternion.identity, q, 1.5f);
// like q=q*1.5, if that were legal
q=Quaternion.LerpUnclamped(Quaternion.identity, q, -0.1f);
// like q=q*-0.1
```

A semi-realistic example, I want to turn to look at the ball, but with an overshoot past it. If I have to turn left, I'll go a little extra left; but if I need to tilt up, I'll go a little extra above it. The plan is to get the rotation to face the ball, then compute 110% of the me-to-there rotation:

```
Quaternion qToBall=Quaternion.LookRotation
    (ball.position-transform.position);

 // the overshoot rotation, starting from my current facing:
Quaternion qtbPast=Quaternion.LerpUnclamped
    (transform.rotation, qToBall, 1.1f);

transform.rotation=qtbPast;
```

To decode the Lerp: 1.0 for the percent gives the ending spot, which is looking directly at the ball. 1.1 is 10% past. Since the start of the lerp is how we're facing now, it computes the full turn, plus 10% going the same way (it took me a while to figure this out).

Quaternion Lerp has an alternate version, `Quaternion.Slerp` (spherical lerp). They both do the same thing. Lerp is actually a less accurate version that runs faster, especially for big spins. But I've never seen a difference or noticed wrong numbers for Lerp.

To summarize:

For points, MoveTowards will handle most things. The speed is in units, which is good for real motion. Changing the speed as it moves can make most effects you need.

Vector3.Lerp is harder to set-up and only useful for when you need a specific time, no matter how far. The me=Lerp(me,end,0.05) version can be fun for special effects.

And these are both short functions with simple math. You can always write it out yourself.

For rotations, Quaternion.RotateTowards handles most things – gives you a smooth, constant shortest-way spin, in degrees/second. Quaternion Lerp is very useful for math – taking half a rotation. Plus making a rare time-based spin.

Vector3.RotateTowards is for special purposes, and awkward to use. You usually move points instead of rotating arrows. And you usually rotate arrows using your rotation. But if all you have is the ending arrow, I guess it's fine.