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.

We know how to move simple rotations – spinning around y. There are more 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 unreliable. It turns out the best way is to set speed-per-second, adding the correct fraction a little at a time.

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

So what we do is forget about frame-rate and updates per second. Instead we time how long each update takes. If an update takes 1/10th of a second, we add 1/10th of the movement. If the next update takes 1/60th, we add 1/60th.

Since we have no way of knowing how long the current update will take, we use the time passed since the last one. At the start of each update, Unity computes that and saves it in `Time.deltaTime`. A typical value is 0.167 (which is 1/60th).

This code adds 3.5 to x every second:

```
public float x;

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

This moves us forward at 3.5 units/second. In our minds `transform.forward*3.5f;` is our speed for second, and `*Time.deltaTime` is the trick to add it a little at a time:

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

This next code moves the red cube towards us at whatever speed-per-second we enter:

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

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

  // standard "add correct fraction of the move arrow":
  redCube.position += toUs*Time.deltaTime;
}
```

An interesting thing – when it arrives it will buzz back and forth. It overshoots, then come back towards us and overshoots again.

6.2 MoveTowards

Unity has a helpful function that does the same thing as that last example. As a bonus, it won't overshoot. The form is `MoveTowards(startPoint, endPoint, moveAmount);`. For example, this moves the red ball towards us by 2 units:

```
redBall.position = Vector3.MoveTowards(
    redBall.position, transform.position, 2);
```

Like any normal function, it doesn't actually move anything. It computes the position. But the form we use is almost always `A=MoveTowards(A,target,amt);`, which moves A. The distance is in real units, not adjusted for anything. To move at 2 units/second we need the deltaTime trick:

```
// 2 units/second (this would be in Update):
redBall.position = Vector3.MoveTowards(
    redBall.position, transform.position, 2*Time.deltaTime);
```

The most fun thing above MoveTowards is how we can easily write it using the vector math we already know. It's a fun review:

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

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

  // standard unit vectors to go a set distance:
  Vector3 unitArrow = AtoB.normalized;
  return A+unitArrow*dist;
}
```

There's nothing wrong with using MoveTowards as a shortcut. It's a function, with a nice name, for a common calculation. But if we need it to work a little differently, or are fighting with it, we can always go back to the basics.

MoveTowards movement can seem robotic, but only if we use the same speed all the time. We can easily change it. This sets the speed to 1/2 the distance to the target, per second, but at least 1:

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

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

Or we could start at speed 0, increasing by 3 per second, to a maximum of 8. Whoever sets new targets would should also reset speed to 0:

```
  speed += 3.0f*Time.deltaTime; // speed goes up 3/second
  if(speed>8) speed=8;

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

MoveTowards is good when:

- We want to go is a specific point. We often just want to move in a direction., which is easier with simple vectors.
- Speed is in units/second. Sometimes we want percent-based movement (like we did in the early chapters).
- We don't want to overshoot. Sometimes the target is an aiming guide, and we want to go through it and past. MoveTowards always stops.

- We want to change the target mid-move. MoveTowards handles that nicely – it move it towards the new target from where-ever it is now.

There are lots of ways to move. MoveTowards does a nice job handling one way: "speed per second to a point".

6.2.1 Lerp

`Lerp` stands for linear interpolation, which is the math way to say "a spot between 2 points based on a smooth 0-1". `C=Vector3.Lerp(A,B,0.25f);` sets C to a spot 25% between A and B, going along the line between them.

You may have noticed that we could do that back in the first chapter. Lerp is very simple. Written out:

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

A practical Lerp example is picking a random spot on a line. We give the end points and a random 0-1:

```
// Random.value creates a random 0 to 1
Vector3 pos = Vector3.Lerp(p1, p2, Random.value);
```

Lerp is a nice 1-line way to say things like "10% of the way from A to B".

Lerp isn't really a movement command, but it's sometimes used that way. This will move the green cube from us to the red one over 2.5 seconds:

```
float pct0to1=0;

void Update() {
  // this takes 2.5 seconds to go from 0 to 1:
  pct0to1 += 1 / 2.5f * Time.deltaTime;
  if(pct0to1>1) pct0to1=0;

greenCube.position =
   Vector3.Lerp(transform.position, redCube.position, pct0to1);
}
```

Notice how it's up to us to gradually move `pct0to1`. We can call the same MoveTowards over and over, and it will work. But Lerp only works if we increase the last input.

There's a common Lerp hack to get a zingy fast-then-slow movement. This zips a cherry, from whereever it is now, to a fixed point. Notice how cherry is the starting spot, and the thing we move:

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

It says to move yourself 5% of the way each time, which is less as you get closer. It looks very zippy. 5% doesn't seem like much, but it adds up. You can't control the speed or the time it takes. This trick is really only useful for quick visual effects.

6.3 Gradually rotating

So far, we know how to gradually rotate by changing the numbers inside of a `Quaternion.Euler(x,y,z)`. There are more useful ways to spin gradually. We can spin arrows into other arrows, spin quaternions into other quaternions, or take fractions of quaternions.

6.3.1 Spinning an arrow

We can rotate an arrow without needing to make a quaternion or use degrees. Give it the final arrow and gradually spin to match. Like most arrow math, you have to imagine both arrows coming from the same point.

The basic code is like a MoveTowards. The third input is the speed, in radians per second(yikes!), and we'll ignore the 4th input for now:

```
  // gradually spins arrow until it's the same direction as endArrow:
  arrow = Vector3.RotateTowards(arrow, endArrow, 0.02f, 0);
```

The really cool thing is how it computes the best way to rotate. Depending on how endArrow aims, `arrow` might spin right, up, or diagonal. It makes a straight spin in whichever direction is best.

Here's an example showing it in use. The 'A' key puts the ball 4 units in front of us. That could be anywhere. Then it automatically spins that arrow to 4 units straight up:

```
public Transform redCube;
Vector3 arrow; // spins from forward*4 to (0,4,0)

void Update() {
  // 'A' key resets and begins new move:
  if(Input.GetKeyDown(KeyCode.A)) arrow=transform.forward*4;

  arrow=Vector3.RotateTowards(
      arrow, Vector3.up*4, 1.5f*Time.deltaTime, 0);
  // NOTE: 1.5 radians is about 90 degrees
  redCube.position = transform.position+arrow;
}
```

If you press 'A', you'll see the red cube is on a curved path. It's at the tip of a rotating arrow. If it doesn't have far to move, the curve is small, but it's still there.

The 4th input is how fast the arrow changes length. If the target arrow is longer or shorter than you are, you can shrink and grow to match it. 0 means not to change length. That seems pretty cool, but it's a pain since they don't auto-synch. You'll probably finish the rotation, then grow in place until you're long enough. Or vice-versa.

6.3.2 Moving a rotation

Rotations have a similar move-towards-like function. You give it the current rotation, the target rotation, 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 aiming straight up, at 60 degrees/second. Try it with starting in various odd directions. It rotates us the closest way, along a nice, straight curve.

It also does something lines don't – it matches the z-roll. If you're on your back, this will roll you head-up. The degrees per second counts z. If you're aimed mostly the correct way but have the exact opposite z-roll, this will take several seconds as if rolls you over.

Here's a practical example. We want to look at the red cube by smoothly spinning. We get the rotation to it and smoothly spin ourself to match it:

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

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

LookAt would make us always face red. This will have us track it, with the possibility it could “out run” us for a while.

This next one is really cute. Pretend our main game is walking around, spinning only on y. We move by changing the float `yFacing`. But sometimes we get knocked down. When that happens we activate our rigidbody, which lets it fall and roll around.

To get up, we turn our rigidbody back off and use RotateTowards to get up:

```
float yFacing; // our normal y-spin
bool isKnockedDown=false;

void getUp() { // call this every frame we're knocked down:
  // compute the standing up rotation:
  Vector3 qWant=Quaternion.Euler(0,yFacing,0);

  // spin us towards the standing rotation:
  transform.rotation=Quaternion.RotateTowards(
      transform.rotation, qWant, 90*Time.deltaTime);

  check if done:
  if(transform.up.y>0.99f) isKnockedDown=false;
}
```

The last line is a trick to check for standing. transform.up is (0,1,0) when we’re perfectly straight. If we’re the least bit crooked in any way, the y-value will be less than 1.

It looks a little fake, like we have some sort of gyro guidance system to straighten up. But it’s also really cool how it gets right back up.

6.3.3 Rotation lerp

Rotations have their own Lerp. It works like the one we’ve seen for vectors. This finds the rotation 1/2-way between q1 and q2:

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

For example, this faces us 1/2-way between the red and green cube. If finds the rotation to each and averages them:

```
Quaternion qToRed=
  Quaternion.LookRotation(redCube.position-transform.position);
Quaternion qToGreen=
  Quaternion.LookRotation(greenCube.position-transform.position);

Quaternion qHalf=Quaternion.Lerp(redCube,greenCube,0.5f);
transform.rotation = qHalf;
```

That example isn't super useful, since we can already look between 2 things by finding the point in-between them.

Lerp lets us cut an angle in half, which we couldn't do before. The secret is to start with Quaternion.identity. We're averaging ourself with 000:

```
// qHalf is 1/2 of q:
Quaternion qHalf=Quaternion.Lerp(Quaternion.identity, q, 0.5f);

// this is like q/4:
Quaternion qHalf=Quaternion.Lerp(Quaternion.identity, q, 0.25f);
```

We can also use the Lerp hack to make a fast-then-slow rotation:

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

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

Notice how it's the same form: we're the starting point, and we also move, and we always move 3% of the way.

This can be nice for transitions. Instead of having the camera's angle snap somewhere, this makes it very quickly aim.

6.4 Misc, summary

Vector3.MoveTowards is the main, easiest way to move points in straight lines. Playing with the 3rd `speed` input can make it look pretty nice. Vector3.Lerp is for math. For when you think "I wonder where 30% of the way between A and B is"?

But both are just shortcuts for things we can already do.

Quaternion.RotateTowards is new, and very useful. It gives a perfect spin to any direction, in degrees per second. handles most things – gives you a smooth, constant shortest-way spin, in degrees/second.

Vector3.RotateTowards is about the same thing. Rotations and directions are similar, after all, and you can turn one into the other.

Quaternion.Lerp is useful for math – taking half a rotation. We have no other way to do things like that.

Both Lerps (points and rotations) have an unclamped version, which means the percent can be 1.1 or -0.5. Rotations with this can be useful:

```
// 1.5 times the rotation:
q=Quaternion.LerpUnclamped(Quaternion.identity, q, 1.5);
// the opposite of the rotation:
q=Quaternion.LerpUnclamped(Quaternion.identity, q, -1);
```

Quaternion Lerp has an alternate version, `Quaternion.Slerp` (spherical lerp). They both do the same thing. Supposedly Lerp is faster, but slightly less accurate. I've never seen a difference.