Chapter 6

Moving

Unity has some built-in commands to help gradually move, or spin. Some are things we couldn't do before, but most are just shortcuts.

6.1 Moving a point

We can already use offsets and normalization to move at a certain speed. Unity has some built-ins to help with this.

First a review. This uses our old unit arrows and distance to move us at a constant rate towards some other object:

```
public Transform goHere;
public float moveSpeed=2.0f;

void Update() {
  Vector3 toTarget = goHere.position - transform.position;
  Vector3 unitToTarg = toTarget.normalized;
  float targDist = toTarget.magnitude;
  float mvAmt = moveSpeed*Time.deltaTime; // units to move this frame

  Vector3 newPos;
  // don't overshoot:
  if(mvAmt>=targDist) newPos=goHere.position;
  else {
    newPos=transform.position + unitToTarg*mvAmt;
  }
  transform.position = newPos;
}
```

The line in the `else` is where most of the work is. We have a unit (length 1) vector to our target. We scale that arrow by how much we want to move (`mvAmt`) and add it.

Before I moved some other cube. This code is moving ourself. It works the same but notice the new way it avoids an overshoot. If we're closer than 1 move, we just place ourself on the target.

6.1.1 MoveTowards

`MoveTowards` is a shortcut for most of that. It takes where you are, where you want to go, and how much to move. Here's a rewrite of the code above using MoveTowards:

```
public Transform goHere;
public float moveSpeed=2.0f;

void Update() {
  float mvAmt = moveSpeed*Time.deltaTime; // units to move this frame

  Vector3 newPos;
  newPos = Vector3.MoveTowards(transform.position, goHere.position, mvAmt);

  transform.position = newPos;
}
```

Obviously, the inside of `MoveTowards` is all the math in the first version, including "don't overshoot."

Because of the no-overshoot part, it's safe to run `MoveTowards` every frame, no matter what. Once you reach the target, `MoveTowards` does nothing – not even any jiggle. Another part of the program can move your target, doing no other work, and you'll start moving that way.

The command looks so neat that it's easy to forget it's still just a 1-time small move. Like anything else, it has to be in Update if you want it to keep moving you each frame.

The third input is the amount to move. The easiest thing to do is to keep it constant, which can be robotic-looking. You can easily change the speed as it moves to get more realism, or to make it prettier.

This example uses a simple equation to make it go faster when you're further away. The numbers aren't important, just that they're based on distance. We'd add these new lines at the start of Update:

```
moveSpeed = ((goHere.position - transform.position).magnitude)/3;
moveSpeed = Mathf.Clamp(moveSpeed, 1, 4);
```

Previously, it moved at a constant 2/second. Now it moves twice as fast when it's far away (the equation has 12+ units away be max speed) and slows, down to 1, as it gets closer. Sort of like a rubber-band effect.

We could do the opposite and have it start slow, quickly increasing to a speed of 2:

```
moveSpeed=0.3f;

// each frame, try to get up to full speed:
if(moveSpeed<2.0f) moveSpeed+=1.0f*Time.deltaTime;
```

When we reach the target, or change to a new one, we'd could reset the speed to 0.3, causing us to accelerate when we start moving again.

6.1.2 Lerp

Another common way to move is what we've been doing with an all-the-way arrow and a 0-1 percent. The technical term for taking a 0-1 percent between two points is Linear Interpolation, shortened to "lerp."

Unlike MoveTowards, we have to know the start point. When we moved a green cube towards a red cube the start point was always us. If we move ourself, we have to save the start point:

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

This code uses math we already know to move us to the target:

```
public Transform goHere;
public float secondsPerTrip=3.0f;

Vector3 savedStart; // where I started my move
float tripPercent=99; // 0 to 1. 99=not moving

void beginMove() {
  savedStart = transform.position;
  tripPercent=0.0f
}

void performMove() {
  // move pct from 0 to 1:
  tripPercent += Time.deltaTime/secondsPerTrip;
  if(tripPercent>=1) { tripPercent=1; }

  Vector3 wholeMove = goHere.position - savedStart;
```

```
  transform.position = savedStart + wholeMove*tripPercent; // <- key line
}

void Update() {
  if(tripPercent<1) performMove();
  else if(Input.GetKeyDown(" ")) beginMove();
}
```

The inside of `performMove` should look very familiar. The rest is to set it up. Unlike `MoveTowards`, the target can't move (if it does, we'll snap to the same spot on the new between-arrow.) We need to reset the start and percent each time.

The built-in Lerp function replaces three lines in performMove:

```
  tripPercent += Time.deltaTime/secondsPerTrip;
  transform.position = Vector3.Lerp(savedStart, goHere.position, tripPercent);
```

It computes the line, makes sure percent can't go past 1, and uses the percent to find the point.

The big difference is moving with a percent takes the same speed to go there. If the points are nearby, it crawls, taking 3 seconds. If they're far apart, it zooms to take 3 seconds.

We could change that by moving the percent slower for further apart objects, but that's a pain. MoveTowards is usually easier.

There's another, completely different way people use Lerp. Each frame you move a fixed percent of the total distance closer. It gives a fast-then-slow movement. It's totally unrealistic, but looks cool and is easy to do.

Here's how the trick looks with regular math:

```
public Transform goHere;

void Update() {
  Vector3 toTarget = goHere.position - transform.position;
  transform.position = transform.position + toTarget*0.05f;
}
```

Each frame it recomputes the new arrow to the target, and moves us 5% of the way along it. We never actually reach it, but we quickly get so close it won't matter.

Here it is written using a `Lerp`:

```
public Transform goHere;

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

When you see the first input matching where you assign it, it's this fast-then-slow method.

6.2 Gradually rotating

Our previous method of making a moving rotation was hand-moving the degrees and remaking the quaternion. A better way is directly rotating a quaternion, using special functions.

This lets us get smooth motion, from any angle to any other, even using two rotations that were made different ways. It's the main reason we like quaternions so much.

6.2.1 Spinning an arrow

Unity has a fun shortcut for spinning an arrow. It works like `MoveToward`. You give it the current arrow, the new arrow, and how much you want to spin this frame.

This example puts a ball to our right, and spins it to point straight up:

```
public Transform ball;
Vector3 toBall = Vector3.right*4; // this is what we will be spinning

void Update () {
  toBall=Vector3.RotateTowards(toBall, Vector3.up*4, 1.5f*Time.deltaTime, 0);
  ball.position = transform.position+toBall

  // reset, for testing:
  if(Input.GetKeyDown(KeyCode.A)) toBall=Vector3.right*4;
}
```

You should be able to see it's not just moving from right to up – it's really spinning the arrow, making the ball move in a curve (if we wanted straight movement we'd use MoveTowards.)

The third input is the spin amount. It's in *radians*, not degrees. 90 degrees is 1.57 radians. That's why it has a speed of 1.5 but goes so fast – that means to rotate almost 90 degrees per second. I'm not sure why it uses radians. 'Real" math uses radians for all angles, but Unity uses degrees to everything else.

`RotateTowards` can also change the length of the arrow. In this case, the starting and ending arrows are both length 4, but they didn't have to be. The 4th input is how fast it grows/shrink the arrow. As soon as it gets to the correct length, it stops. If it doesn't change size, the 4th input won't matter. I think 0 looks good for those cases.

If your math is wrong, the rotation and size change will finish at different times. It could rotate all the way then finish shrinking, or shrink to the final size halfway through the rotation. For example:

```
toBall = Vector3.right*2;
  ...
toBall=Vector3.RotateTowards(toBall, Vector3.up*4, 1.5f*Time.deltaTime, Time.deltaTime);
```

For about a second this spins to face up while shrinking to length 3. Then, while pointing up, takes another second to shrink down to length 2. Doubling the 4th input would fix that.

6.2.2 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(0,90,0);

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

This spins us from our current facing to due east, spinning at 60 degrees/second.

This (and RotateTowards) takes the most direct line, and has no problem going diagonal. You could be facing any direction and this will rotate you by the most direct route.

Since these are rotations, it also takes into account the z-roll. This will just spin us in place from head up to head down:

```
// start exactly forward:
void Start() { transform.rotation = Quaternion.identity;}

void Update() {
  Vector3 qWant=Quaternion.Euler(0,0,-180);
  transform.rotation=Quaternion.RotateTowards(transform.rotation, qWant, 30*Time.deltaTime);
}
```

We can also do both at once. This flips us on our back while slowly tilting us a little bit up:

```
// start exactly forward:
void Start() { transform.rotation = Quaternion.identity;}
```

```
void Update() {
  Vector3 qWant=Quaternion.Euler(-10,0,-180);
  transform.rotation=Quaternion.RotateTowards(transform.rotation, qWant, 30*Time.deltaTime);
}
```

These always both finish at once, since they count as one big rotation. This one quickly rolls 180 on z while slowly tilting the 10 on x.

6.2.3 Rotation lerp

Rotations have there 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. This is because `q*0.5` or `(q1+q2)/2` don't work at all.

To get a rotation 1/2-way between `q1` and `q1`, 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);
```

Both of the old moving-Lerp tricks work.
`Quaternion.Lerp(startRotation, endRotation, pct0to1);` will spin you from one to the other as the percent goes to 1. RotateTowards is probably simpler.

The percent-based Lerp will give a fast-then-slow spin. This zooms us to facing up, easing into it as we get near:

```
void Update() {
    transform.rotation=Quaternion.Lerp(transform.position, Vector3.up, 0.03f);
  }
}
```

There's an alternate version named `Quaternion.Slerp`. It does the same thing as Lerp – you can use either one. Slerp uses a better but slightly slower math equation, but I've never seen a difference.

Like vector Lerp, Quaternion Lerp only uses 0-1 percents. If you really need 150% of your angle, `LerpUnclamped` lets you use any percent. Exs:

```
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
```

This can solve some very specific math problems, like needing to double an angle (use 2 for the percent,) or find the opposite angle (use -1.)

6.3 Time.deltaTime

This is a common trick that lets us use nice speeds for movement. The problem is we prefer to think of movement as speed-per-second, but these commands don't work that way. `Time.deltaTime` is a way to fix that.

I'll show you how to use it first, then the explanation:

Whenever you run a move command every frame, the easiest thing is to use amount-per-second for the speed. Use the per-second movement, multiplied by `Time.deltaTime`. This moves at 5 units/second:

```
Vector v1 = Vector3.MoveTowards(v1, target, 5*Time.deltaTime);
```

If we run this every Update, it moves a little each time, adding up to 5 each second.

The actual value of `Time.deltaTime` is how many seconds it's been since the last time Update ran, which is always something tiny like 0.0166 seconds. You can remember it since *delta*, in math, means how much something changed. It's the change in time.

To see how it works, suppose you have exactly 60 evenly-spaced Updates every second. `Time.deltaTime` will always be 1/60th – 0.0166. Each Update you'll be adding 1/60th of 5, which is what we want to add to 5/second.

Next suppose one frame gets skipped. The next Update will have double the `deltaTime`. We'll move twice as much that update, making up for the time we missed.

For rotations, this spins at 60 degrees per second because of the 3rd input:

```
Quaternion qq = Quaternion.RotateTowards(qq, qWant, 60*Time.deltaTime);
```

If we wanted to make it spin faster, increasing at a rate of 40 degree per second, we'd use `Time.deltaTime` for the speed increase:

```
// this adds 40 every second:
rSpeed += 40*Time.deltaTime;

Quaternion qq=Quaternion.RotateTowards(qq,qWant, rSpeed*Time.deltaTime);
```

When you see things using `Time.deltaTime`, it's easiest to not think of it as a variable. Look at what's next to it, and that's speed/second.

One last note, it's still just a 1-time thing. The trick works because we do it every Update. For example, this would move us a tenth of a unit each time we pressed "a," and no more:

```
if(Input.GetKeyDown("a"))
  transform.position = Vector3.MoveTowards(transform.position, dest, 6*Time.deltaTime);
```

But if we changed it to `GetKey`, which is true as long as the key is held, it would run every Update and would give a speed of 6/second.