Chapter 5

Combining rotations

Instead of thinking of a rotation as an absolute facing, you can think of it more like an offset: a change from where you're looking now.

We can apply a rotation to any arrow, to spin it that much. Or we can apply a rotation to another rotation, to rotate the rotation. For example: face ourselves somewhere, then add an extra twist.

This is some of the hardest stuff, but if you have the basic idea, you can usually trial and error and figure out what you need after some testing.

5.1 Rotate an arrow

For rotations, we repurposed `*` to mean "apply this rotation." If `v1` is an arrow and `q` is a rotation, `q*v1` is that arrow rotated by that much.

Rotating a forward arrow is the simplest way to use it. This creates a 20 degree rotation on y, then applies it to a forward arrow. It gives us a 20-degree arrow:

```
Quaternion spin = Quaternion.Euler(0,20,0); // 20 degree y-spin
Vector3 v = spin*Vector3.forward; // tilted arrow
```

Using `Vector3.forward` is special. +z is the all-zero rotation, so a forward arrow counts as a no-rotation arrow. `spin*Vector3.forward` turns any rotation into it's arrow.

That's a useful thing to know, but the real use of rotation-times-vector is being able to use it on any vector. For example:

```
Vector3 v1 = spin*Vector3.right;
Vector3 v2 = spin*transform.right;
```

`v1` will be your right arrow, spun an extra 20 degrees. It's merely an arrow pointing at 110 degrees, but we made it in this cool way. With `v2` we don't

know which way it's facing, but it's your local +x arrow spun an extra 20 degrees around y.

A fun use is spinning an object in a circle. We can take any arrow and hit it with a gradually increasing rotation. This takes a forward arrow and slowly spins it 0-180 degrees, around us:

```
public Transform redCube;
float degrees=0;

void Update() {
  Quaternion spin = Quaternion.Euler(0, degrees, 0); // y-spin
  Vector3 arrow = spin*(Vector3.forward*2); // <- the key line

  redCube.position = transform.position + arrow;

  degrees+=3; if(degrees>180) degrees=0;
}
```

I had it snap back at 180 so you can clearly see the starting position. This will start north, trace out a half circle going clockwise, then snap back to north.

The extra `*2` is for contrast. Remember `Vector3.forward*2` is merely (0,0,2) – a length two forward arrow. The `spin*(Vector3.forward*2)` applies the spin to the one long arrow.

We could instead start with an arrow pointing some other direction. Replacing the line with this next one would start the half-circle on the right, tracing a clock-wise half-circle left and down:

```
  // half-circle from right, going down and left:
  Vector3 arrow = spin*(Vector3.right*2);
```

Or we could start with just any line. This starts with a mostly-left arrow, tracing out mostly the top half of a circle:

```
  // half-circle on top, cocked a little:
  Vector3 arrow = spin*(new Vector3(-4,0,1)) ;
```

This arrow is longer. The equation rotates the actual arrow, so this long arrow stays long as it rotates.

Some notes on the rules for using these:

- The rotation has to come first, then the vector second. You'd think `v*spin` would also work, but it's just an error. It has to be `spin*v`.

- The star(`*`) isn't really a multiply. It's being re-used here as the "apply a rotation" symbol. It's a little like how `"cow"+"bell"` knows the `+` isn't a regular math plus.

  We're allowed to repurpose math symbols – it's called Operator Overloading.

- The regular math rules apply, so `v1+spin*v2` spins `v2` first, then adds `v1`. Using `spin*(v1+v2)` adds the arrows first, then spins the result.

We can spin using any axis we want. Going around the x-axis gives an edge-on vertical half-circle:

```
// bottom half of a circle around the x-axis:
Quaternion spin = Quaternion.Euler(degrees, 0, 0);
Vector3 arrow = spin*(Vector3.forward*2);
```

It's the bottom half because of the left-handed-rule. We start forward, and spin down. If we used `Vector3.up` we'd get the north half of a circle.

Spinning a point on the axis does nothing. Either of these two lines won't make it move at all:

```
// spinning up around y does nothing:
Vector3 arrow = Quaternion.Euler(0, degrees, 0) * Vector3.up;

// spinning right around x does nothing:
Vector3 arrow = Quaternion.Euler(degrees, 0, 0) * Vector3.right;
```

In the first one, the up-arrow really does spin around y, but it just twirls in place. Points don't have a facing, so it stays as (0,1,0) no matter how much we spin. Spinning a right or left arrow around x is the same thing – no change.

The other oddball is spinning points that aren't "on" the circle. We can take a point 5 up and 1 over, and spin it around y. It will make a tight radius 1 circle, 5 units up. In other words, the 5 up won't change. Only the 1 sideways part will spin:

```
// makes a radius-1 half-circle, 5 units above us:
Vector3 arrow = Quaternion.Euler(0, degrees, 0) * (new Vector3(1,5,0));
```

Sometimes it helps to stop imagining it as an arrow, and instead as a point, glued into a 3D grid. Then imagine the whole thing spinning. You'd see the dot at (1,5,0) up in the air, making a little ring around the y-axis.

So far we've been making rotations with the `Quaternion.Euler` method, but any way to make a rotation will work. Suppose we're angled funny and want to orbit something from our right to left, which is around our z-axis.

Using `transform.right` for the starting arrow seems right. To make the spin be around our local z, we can use angleAxis:

```
// half-circle around our local +z:
Quaternion spin = Quaternion.AngleAxis(degrees, transform.forward);
Vector3 arrow = spin*(transform.right*2);
```

To sum up:

- Spinning a point which is on the axis won't change it. For example, spinning a forward arrow around +z.
- Use the forward arrow to turn a rotation into it's arrow.
- For every other arrow, `q*arrow` works like an offset, spinning it more in that direction.
- Any kind of rotation can be used: one's made with `Quaternion.Euler`, or `AngleAxis` or our `transform.rotation` .... No matter how you get one, all rotations are quaternions, so can be used for this trick.

Fun fact: the computer calculates `transform.forward` using `transform.rotation*Vector3.forward`. It bends the forward arrow by your rotation.

`tranform.right` is the same idea. It's `transform.rotation*Vector3.right`.

Here's one longer example. I'd like to shoot a ball randomly forward, in a cone. My idea is to use two steps. First take the forward arrow and cock it right 0 to 10 degrees. Then spin that arrow around z by a random 0-360:

```
Vector3 dirArrow = Vector3.forward;
// small rightward cock:
dirArrow = Quaternion.Euler(0, Random.Range(0,11),0) * dirArrow;
// random 0-360 z-spin:
dirArrow = Quaternion.Euler(0, 0, Random.Range(0,360)) * dirArrow;
```

After those two spins, dirArrow is somewhere in a 10-degree forward cone. Shooting a ball that way is a simple use of the `dirArrow` arrow.

Sometimes we want the first few shots to miss. It's easy to put a hole in the middle by changing the first angle to be 5-10. Our almost-forward arrow will be somewhere in a donut.

5.2 Combining rotations

We also use the star symbol to combine rotations. First a simple example. We know (-45,90,0) is really done in two steps, y then x. We can make each of those separately and combine them:

```
Quaternion ySpin90 = Quaternion.Euler(0,90,0);
Quaternion xSpin45 = Quaternion.Euler(-45,0,0);

Quaternion y90thenx45 = ySpin90*xSpin45; // <-- the new line

transform.rotation = y90thenx45;
```

The rule is that the second rotation uses the new local axes. In `ySpin90*xSpin45`, the second part tilts us up on *local* x.

If you flipped them, `xSpin45*ySpin90`, you'd get a different direction. We'd tilt up on the real x, then we'd spin 90 degrees on the local y, putting us back to the ground facing exactly right, with a 45 degree roll.

It seems seriously weird that the order matters, but that's really the way it is. It's not even a quaternion thing. The way rotations are, the order matters.

Here's a real example, I want to look at something, then roll on my side. Getting a look rotation puts our head up. So we add a 90 degree roll on local z:

```
// look rotation to red cube, head up:
Quaternion qLook = Quaternion.LookRotation(redCube.position-transform.position);
// standard 90 degree z-roll:
Quaternion qzRoll90 = Quaternion.Euler(0,0,90);
// look, then local roll:
transform.rotation = qLook*qzRoll90;
```

We can see `qzRoll90` is just a simple z-roll. Like any offset, what it does depends on how we use it. Since it's second, it automatically applies to the local z.

Just for fun, this looks at something while gradually rolling. It's using the same local z-roll trick, but now with an increasing 0-360:

```
float zRoll=0;

void Update() {
  Quaternion qLook=Quaternion.LookRotation(redCube.position-transform.position);
  Quaternion qRoll = Quaternion.Euler(0,0,zRoll);
  transform.rotation = qLook*zRoll;

  zRoll+=4;
}
```

Here's a trickier one. We want to make an orbit, right-to-left (the z-axis circle we made before) and add a forward/back zig-zag. It should trace out a wavy circle.

The first part will be the same: spin a `Vector3.right` around z. To add the waves, think of the right-facing arrow. Wiggling y will make it wave. As we rotate in the circle, the local y axis rotates with us, staying 90 degrees ahead. Wiggling on local y will always be a perfect forwards/backwards:

```
public Transform ball;

float zSpin=0; // 0 to 360
// The wobble:
float ySpin=0; // goes between -20 and 20
int yDir=+1; // +1 or -1, to make it go back&forth

void Update() {
  // main circle:
  Quaternion qCircle = Quaternion.Euler(0,0,zSpin);

  // back-and-forth y-wobble:
  Quaternion qWobble = Quaternion.Euler(0,ySpin,0);

  // now combine them: orbit spin, local y-wobble
  // ...and apply that to a long Right arrow:
  Vector3 toBall = qCircle*qWobble*(Vector3.right*2); // <- key line
  ball.position = transform.position + toBall;

  // move them:
  zSpin+=2;
  ySpin+=3*yDir;
  if(ySpin>20) { ySpin=20; yDir=-1; }
  else if(ySpin<-20) { ySpin=-20; yDir=+1; }
}
```

Obviously, this is something where you want to add comments explaining the plan. The key line `qCircle*qWobble*Vector3.right` isn't easy to just look at and know what it does.

5.3 Visualizing rotation combinations

I wrote that in `q1*q2` the second rotation uses the local axis of the first. Well ... that's one way to look at it. There are others.

The most important thing is the order matters. If you combine rotations A, B and C, you have to choose between `A*B*C` or `B*A*C` ... 6 combinations. All do different things.

A lot of times, we're thinking of one thing as the base rotation, and the rest as ways to adjust it. Here are the ways to think of it:

- `base*local`. Multiplying after the starting spin applies the new one as local.
- `world*base`. Multiplying before the starting spin applies the new one using world axes.

- Left-to-right `base*local1*local2*local3`. Each one after the first uses the local axis so far.
- Right-to-left `world3*world2*world1*base`. You can think of the right one as the base spin, and then go right-to-left and apply each one using world coordinates.

That seems really confusing, that `A*B*C` could mean left-to-right local, or right-to-left world. For real, you start with a plan. If your plan uses local rotations, add them to the back. If it's global rotations, add them to the front.

In case you were wondering, this is the way real rotation math works. Unity is just copying it, and not making it any more complicated than it was already.

Examples:

This is the cone with a hole in it example, rewritten to think in local axis. Assume we want to shoot the ball in almost our local forward. First we take real forwards and spin to face my forward. The next step is rolling 0-360 on local z. We'll still be facing forward, but the local y-axis will be in a random direction. Then we can cock 10-20 degrees on local y.

In other words, point your finger forward, spin randomly in place, and cock it a little sideways. Here's the code:

```
// 360 roll on z, for the "miss direction":
float missDir = Random.Range(0,360);
Quaternion qCone = Quaternion.Euler(0,0,missDir);

// sideways cock 10-20 degrees:
float missAngle = Random.Range(10.0f, 20.0f);
Quaternion qCock = Quaternion.Euler(0, missAngle, 0);

// combine them into the final direction:
Quaternion qShoot = transform.rotation*qSpin*qCock;

ball.position = transform.position+qShoot*(Vector3.forward*2);
```

That second-to-last line is thinking in local, right to left: face my way, random roll on local z, cock using randomly-facing local y. In case you were wondering.

Because of the rules, we can also think of this right-to-left as world: small y-rotate, random 360 spin on world z, then apply the world rotation to get my facing. But I think this is easier to imagine as left-to-right local. Applying a series of local rotations sounds more complicated, but it's more natural.

Here's an easier one using the world trick. We want to start just any facing, keep it, and have the script spin me around global y. We can do that by putting the y-spin first:

```
Quaternion startSpin; // saved value
float yDegs=0;

void Start() { startSpin=transform.rotation; }

void Update() {
  yDegs+=3;

  Quaternion ySpin = Quaternion.Euler(0,yDegs,0);
  transform.rotation = ySpin*startSpin; // <- key line, ySpin is first = global
}
```

Read the last line as: `startSpin` is the base, with world `ySpin` done to it.

Let me just say again this is some of the hardest stuff to use, but no one is going to know you took a few tries to get something working. The keys are: remember `q1*q2` is different from `q2*q1` for rotations. For each rotation decide if you want it on global, or local so far, and put it in front or in back, depending. Remember you can mix&match both types. And write down your plan in a comment, since the line combining them will make no sense without it.