Chapter 5

Combining rotations

Instead of thinking of a rotation as an absolute facing, you can think of it as a change. In other words, `Quaternion.Euler(0,90,0)` could mean facing east, or it could mean a quarter circle spin from where-ever you are now.

We can apply a rotation to an arrow, which spins the arrow. Or we can apply a rotation to another rotation, which adds them together, sort of.

Rotations are applied using a re-purposed star: `v1=q*v1;` rotates `v1` by `q`. It's a little like how `"cat"+"fish"` uses `+`. We're not actually multiplying the rotation numbers together, but mathematicians think `q1*q2` is a natural way to say "combine these rotations".

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 Rotating an arrow

To apply a rotation to an arrow, use `rotation*arrow`. For example, this spins a right-pointing arrow by 90 degrees:

```
Vector3 vf = new Vector3(3,0,0); // long right arrow
// standard 90 degrees spin:
Quaternion y90 = Quaternion.Euler(0,90,0);

vf = y90*vf;
```

The trick is to think of `y90` as a free-floating 90-degree spin. Applying that to a right arrow turns it into a backwards arrow.

It works with any kind of spin and any arrow. This finds an arrow to the red cube, then spins it 20 degrees:

```
Vector3 toRed = redCube.position - transform.position;
Quaternion y20 = Quaternion.Euler(0,20,0);

Vector3 almostToRed = y20*toRed;
// place a ball there to prove we did it:
theBall.position = transform.position + almostToRed;
```

The ball will be the same distance from us as the red cube – we only rotated the arrow. It will be at the same height, since that's how y-spins work.

Way back in the game board example I said we only needed two corners on the bottom, not even lined up, and we could compute the other two. To do that we'll take the arrow across the bottom and spin it backwards 90 degrees to make the arrow along the side:

```
public Vector3 lowerLeft, lowerRight; // user enters these two

  Vector3 acrossArrow = lowerRight-lowerLeft;

  // the new part to spin the board edge arrow:
  Quaternion spin90back = Quaternion.Euler(0,-90,0);
  Vector3 forwardArrow = spin90back*acrossArrow;
```

The computed `forwardArrow` is the same length as the arrow along the bottom, running at 90 degrees to it. Adding it to each of the bottom corners will give us the top two.

A fun trick is hitting any arrow with a changing rotation. The arrow will spin around, the tip tracing a circle. This spins a red cube in a half-circle 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);

  redCube.position = transform.position + arrow;

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

It only goes 180 then snaps back on purpose, so we can see where it starts. Since we're angling a forward arrow, this traces out the right half of a circle.

If we used `Vector3.right*2`, we'd start there and spin over the bottom half.

Some notes on the rules for using these:

- You can't flip the order. `spin*v` rotates `v`, but `v*spin` is an error. That's the rule from real math.
- The star(`*`) isn't a multiply. For example, in rotation (10,45,90) times point (3,4,8) we're definitely not taking 10 times 3, 45 times 4 and 90 times 8.

  We're really running a function with those two inputs, doing lots of ugly angle math.
- Regular precedence rules apply. `v1+spin*v2` spins `v2` first, then adds `v1`. Using `spin*(v1+v2)` adds the arrows first, then spins the result.
- There isn't a `q1+v1`. We picked multiply to mean "spin by" and there's nothing else we'd even use `+` to do.

5.1.1 Converting a rotation into its arrow

It's still nice to sometimes think of a rotation as the way it aims you, which is like an arrow. We can use our new math to get that arrow.

The trick is that no rotation, (000), is like a forward arrow. Any other rotation is how it would bend a forward arrow. Pretty sneaky, right?

`Quaternion.Euler(0,45,0)*Vector3.forward` bends the forward arrow 45 degrees, which is the way (0,45,0) counts as facing.

`q*Vector3.forward` converts rotation `q` into it's arrow.

For example, `Quaternion qq = Quatenion.Euler(-30,45,0);` is facing forward, right and a little up. `qq*Vector3.forward` is an arrow aimed forward, right and a little up.

Unity uses this trick to compute `transform.forward`. It's really our rotation times the forward arrow: `transform.rotation*Vector3.forward`.

Before, we knew how to use LookRotation to convert an arrow into a rotation. Now we know how to go the other way.

5.1.2 More arrow rotation

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

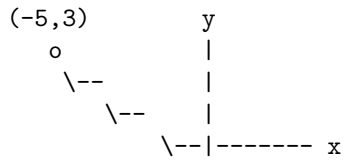
```
float degrees; // pretend this goes from 0 to 180 and snaps back

  // underground 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: +x spins forward and down. If we changed to using a backwards arrow we'd spin over the top half.

Of course, we can code `degrees` to go backwards to spin the other way.

I've been cheating a little by using easy examples. It can be hard to visualize spinning just any arrow around any axis. For fun, let's spin (-5,3,0) a few ways.

First let's spin it around y:



The -5 will spin, making a radius 5 circle around y. The 3 won't change. The whole circle is at height 3. You can imagine the arrow tracing out a shallow upwards-facing cone.

If we roll it forwards, around the x-axis, the -5 stays the same. We'll trace out a radius 3 edge-on circle, always 5 to the left.

You can imagine the arrow, glued to x as it spins, tracing out a longer left-facing cone. But the part we see, the tip, is still just a circle.

Spinning around z is the easiest, since we're at right angles to it. To z, we count as a regular length 5.8 arrow with a funny starting spin. z-rotation will spin a circle right on the picture, centered right there on (000).

Sometimes a trick is to stop thinking of an arrow, and just imagine the point. Draw a line straight to the axis, at the nearest possible spot. It always traces out a circle with that radius.

A funny case is accidentally spinning an arrow around itself. It won't change at all. For example, this spins an up arrow around another up-arrow:

```
// spinning up around y does nothing:
Vector3 arrow = Quaternion.Euler(0, degrees, 0) * Vector3.up;
degrees+=2;
redCube.position = transform.position + arrow;
```

Pretty much this says to point straight up, then spin your finger in place. It's like we're tracing out a circle of radius 0.

It's not an error; just probably a mistake. The red cube will always be 1 unit above us.

So far we've been making rotations with the `Quaternion.Euler` method, but any way to make a rotation will work. Suppose we want to spin a ball around us, from our personal right to left. The spin should be around our forward arrow. AngleAxis was made for funny spins like this:

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

degrees+=2; if(degrees>180) degrees=0;
ball.position = transform.position + arrow; // ball circles us
```

Essentially, we're pointing along our forward, sticking out our thumb to the right, and spinning our hand.

A summary:

- 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.
- There's nothing different about spinning around z. The "z doesn't matter" rule only happens for facings made with y,x,z.
- Spinning arrows that aren't going straight away from the axis is a little tricky. You have to sort of draw another line from the tip, straight to the axis, and spin that. Or else imagine the arrow spinning to make a cone.
- Any kind of rotation can be used. Especially `AngleAxis` to spin something around diagonal lines.

5.1.3 Ball cone-shooting example

Here's one longer example. I'd like to shoot a ball along +z, but randomly 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;
```

Now `dirArrow` is flat, pointing forwards and a little bit right. This next part twirls it around z. Instead of slightly right, it will be slightly up, or left and down .... The possible directions make a narrow forwards cone:

```
// random 0-360 z-spin:
Quaternion zRand360 = Quaternion.Euler(0, 0, Random.Range(0,360)) ;
dirArrow = zRand360 * dirArrow;
```

We can use our usual tricks to place a ball at the end of the arrow, and give it a speed (if we make it a rigidbody) in that direction. We can shoot balls almost-forward.

Finally, suppose we want to shoot the balls the way we're facing. We can apply our rotation to `dirArrow` to spin it that way:

```
Vector dirArrow = transform.rotation*dirArrow;
// aims forwards from us, in a 10 degree cone
```

It's pretty much the trick of turning `Vector3.forward` into the arrow for some rotation. `dirArrow` is almost forward, so the result is almost our forward.

5.2 Combining rotations

We can also apply one rotation to another. It uses the star symbol in the same way. You write it like multiplication, but it's really running rotation math. `q1*q2` combines `q1` and `q2` into one big rotation.

`transform.rotation = q1*q2;` is completely legal.

But we run into the local/global axis problem we had with euler angles. We take rotation `q1` and apply a 30 degree z rotation. Is it on our current z-axis, or the global z axis? Those can be very different.

The good part is, we can pick which one want. The sort of bad part is we pick by which order they go in.

5.2.1 Applying local rotations

When you multiply rotations, the first one is the starting rotation, and the second one is on the local axis of the first.

A simple example, we can take a LookRotation and give it a z-roll (the one on local z, that spins us without changing aim direction):

```
// simple look rotation to red cube:
Vector3 toRed=redCube.position-transform.position;
Quaternion qLook = Quaternion.LookRotation(toRed);

// standard z-roll. Pretend z goes from 0 to 360:
Quaternion zRoll = Quaternion.Euler(0,0,degrees);
```

These are nothing special. Notice how `zRoll` is a perfectly ordinary spin on z. It's not local or global – that depends on how we use it. This applies it as a local to the look rotation:

```
// combine: lookRotation with a z-roll:
transform.rotation = qLook*zRoll;
```

If we let it run, with the degrees on zRoll changing, we'd stayed with our nose aimed at red, z-rolling in place.

For a more math-y example, we know `Quaternion.Euler(45,90,0)` is a shortcut for a spin on global y, then local x. It's a good shortcut, but we can also write it the long way:

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

Quaternion y90thenx45 = ySpin90*xSpin45;

transform.rotation = y90thenx45;
```

That's just a bad way to write `Quaternion.Euler(-45,90,0)`. But we can flip it to have global x then local y.

As we all know, the Mark-II plasma cannon is mounted on a circular base with a sideways rod through it. The whole base tips straight back to aim up and down. The scarey-looking barrel swings side-to-side on the base like a clock-hand, swinging in a tilted curve.

In other words, it's an exact global-x, local-y. The code to aim:

```
float xTilt, ySpin;

void Update() {
  readUserInputs();
  // pretend AD keys move ySpin between -90 to 90 ...
  // ...and WS moves xTilt between 0 to -90 (straight up)

  Quaternion ySpin = Quaternion.Euler(0,ySpin,0);
  Quaternion xSpin = Quaternion.Euler(xSpin,0,0);

  plasmBarrel.rotation = xSpin*ySpin; // <-x is first
}
```

If you try this, you can totally see how y is local. At first it's a normal side-to-side. But if you tilt back to aim high, `ySpin` is a standing side-to-side arch.

If you've ever seen a real Mark-II plasma cannon, you'll recognize the distinctive rotation. You may also realize this way is much better at aiming above us than the yx way. But we just traded – it's horrible at aiming sideways. Degree and axis-based spins always have bad spots.

This next one uses 2 local rotations. We're making a game for kids called Left or Right. A 3D cow will always have a little ball ahead of it and a little bit left or right. On higher stages, the cow can roll on its side, or any which way.

For example, the cow could be lying on it's right side, aimed above the ball. The ball is to the cow's right.

The plan is to put the ball anywhere, then: 1) start with the rotation looking at the ball, 2) add a random 360 z spin. We're still looking at the ball, 3) add a +/-10 degree y-spin. Now the ball is perfectly to the cow's left or right.

The code:

```
  Vector3 toCube = cube.position - cow.position;
```

```
Quaternion lookCube = Quaternion.LookRotation(toCube);

float zz=Random.Range(0.0f, 360.0f);
Quaternion zRand360 = Quaternion.Euler(0,0,zz);

float yy=10; if(Random.value<0.5f) yy*=-1;
Quaternion yLeftRight=Quaternion.Euler(0,yy,0);
```

Those are the three simple angles. Now we combine them according to the plan:

```
cow.rotation = lookCube*zRand360*yLeftRight;
```

Visualizing it is similar to visualizing normal euler rotations: a starting rotation, then two locals; each using the current axis. Imagine the cow aimed, then spun on it's back or something. Be the cow. Then apply the final small left or right spin.

5.2.2 Applying global rotations

It's not as often that we have a starting rotation and want to apply another as a global, but we can do it.

The starting rotation goes second, and the one to apply as global goes first.

Suppose we want the red cube to have my rotation, spun by 90 degrees on the real y:

```
// pretend y90 is euler(0,90,0)
redCube.rotation = y90*transform.rotation;
```

In our minds, `transform.rotation` is the start, with `y90` added to it. Since we put it in front, it's global – a twist of our facing on the real y.

Adding a rotation as global is good when we have a plan that needs it. Here's the plan for the "almost in front cone" using only rotations: face forward and randomly 0-10 degrees right; then spin random 0-360 around z. Spinning around local z does nothing, but global z twirls us in a nice cone:

```
Quaternion qRandCone = qRandZ360 * qRightALittle;
```

The plan says `qRightALittle` is the start. We added `qRandZ360` in front to make it spin us around global z.

We can use the `*Vector3.forward` trick to turn it into an arrow. Or use it directly, for example, to aim us: `transform.rotation=qRandCone;`.

Just in case, here's the code making the rotations:

```
// forward arrow tilted right:
float tiltAmt=Random.Range(0,10.0f);
Quaternion qRightALittle = Quaternion.Euler(0, tiltAmt, 0);

// random any spin around z:
float zz=Random.Range(0, 360.0f);
Quaternion qZrand360=Quaternion.Euler(0,0,zz);
```

Next we want something for a game where a cannon spins in a circle and you have to tap when it's pointing at the target. The target can be higher than us, and we want to get the angle right.

The plan is to use LookRotation to get the correct up/down aim. Then spin around the real y-axis:

```
Quaternion qLook=Quaternion.LookRotation(target.position-transform.position);

// pretend degrees increase 0 to  360:
Quaternion y360=Quaternion.Euler(0, degrees,0);

Quaternion facing=y360*qLook;
```

The last line takes `qLook` as the main facing, then applies `y360` in front to get the global spin.

In our game you win if you tap within 5 degrees of the target, which would be 0-5 or 355-360, since 0 degrees is exactly correct.

5.3 Misc rotation combinations

With the "after=local, before=global" multiplication rule, you may have noticed a possible problem: `q1*q2` could be either. It's `q1` with `q2` applied local. It's also `q2` with `q1` applied global.

You should think of it whichever way makes the most sense to you.

For example, `lookSpin*zRoll` feels like lookSpin comes first. It's easy to visualize as a 3D model aiming, then spinning along it's local z.

But `ySpin*lookSpin` also feels like lookSpin comes first, adding ySpin as global. The 3D model is aiming, like before, but now we're giving it a merry-go-round global spin, which pulls the aim left or right.

`lookSpin*y90` is a fun one. Try thinking of `y90` first, which is an eastward facing. Then `lookSpin` is applied to it as a global. It will twist us to face the lookSpin direction, except we're already facing right. I kind of have an idea of the result, but hmmm ....

Let's try the way starting with `lookSpin`. We're facing the target and apply y90 on our local axis. That's easy – y90 on local is a right turn. All together it says "face the target and turn right". It's the equation for making your left

side face the target.

Here's one more puzzle: imagine rotation `x360` is changing to constantly roll forwards on x. And `ypm20` means "y plus/minus 20". It's a y-spin back and forth from -20 to 20. It's basically facing forward, with a sideways wiggle.

As they both move, what does `x360*ypm20` make?

I'll imagine `ypm20` as the start. It's like pointing two fingers forward in a V – it goes back&forth between them. Then we apply `x360` as a global. That rolls us directly forwards. We'll be somewhere inside that V. I can sort of visualize it, but don't love it.

The other way is `x360` first, with `ypm20` applied as a local. I can understand that – a small local y-spin is looking to your left and right. Applied to a cow, it's a forwards spin, with a left/right swing. Applied to the forwards arrow, it's a circle with a zig-zag (imagine tracking a dot on the cow's nose).

Maybe the first way makes more sense to you. But it's nice how there are two possible ways to visualize, and you only need to understand one.

Suppose someone's notes say that `qRat` is the base, in the rotation equation `qCow*qCat*qRat`. That means the others are applied as global. Go backwards from right to left. Apply `qCat` using global coordinates. If it's 20 on x, roll 20 forward on the real x. Then apply `qCow` as global to that.

It's not common, since not many problems are solved that way. But you can add a chain of global rotations by going right-to-left.

Sometimes you read it from the inside. For example `ySpin*lookSpin*zRoll`. This feels like `lookSpin`, with a local z-roll, then a global y-spin. It's the "wrong aim" example, with a fun extra tilt.

Summary, notes:

- When you see `q1*q2`, think about which way it makes the most sense: q1 with local q2 applied, or q2 with global q1 applied.
- Try to visualize what each rotation means. LookAt rotations are good starting spots, whatever position. Small local y-spins feel like "turn right or left". Likewise local z-spins feel like a fun "spin in place". Applying your rotation as global shifts it to your point-of-view.

  Think about whether a rotation is moving, and how much.
- Every rotation had a plan. In `q1*q2*q3` the person making it may have started with any of those three, applying the rest as global or local tweaks. It probably makes the most sense if you read it using their order.
- You can read these equations left-to-right and they're all locals. Or read them right-to-left and they all count as globals.

One more fun example using these. We want a model to z-roll around it's original facing. First we'll save the start rotation and create two different z-spins: global z, and our local z:

```
Quaternion mySavedQ;
Vector3 mySavedForward;
float degrees=0; // moves 0 to 360

void Start() {
  mySavedQ = transform.rotation;
  mySavedForward = transform.forward;
}

void Update() {
  degrees+=2; if(degrees>360) degrees=0;

  // global z-spin:
  Quaternion z360=Quaternion.Euler(0,0,degrees);

  // a different spin on our z:
  Quaternion zMe360=Quaternion.AngleAxis(degrees, mySavedForward);
```

Plan #1 is the most obvious one: start with our rotation and spin over our personal z. It's a world spin (which seems odd, but it is) so it goes in front:

```
transform.rotation = zMe360*mySavedQ;
```

In theory we can try reading this left-to-right instead. But spinning on `zMe360` before we're lined up on it is just a mess.

The other plan starts with our facing and applies a normal z-spin on our local z. In other words, it's a basic local z-roll. Easy to read:

```
transform.rotation = mySavedQ*z360;
```

For fun, we can sort of read it right-to-left. Start with us facing forward, rolling on z. Then global twist that into our correct rotation, which keeps the same z-roll.

I like this example since the laziest way is also the best. It feels like computing our real angleAxis z should simplify the rest, but it doesn't.

5.4 Z, global x, global y

Our math says we can read standard euler rotations backwards. Instead of y, local x, local z, we can read `y*x*z` as `z` with the other two applied as globals.

For run, let's try it:

Imagine facing forwards, spinning on z. Next global x angles us up or down. The z-roll gets carried along with us.

Finally we spin on the real `y`. The z-roll and the upward tilt are carried along as we swivel left/right.

It's bizarre that those two ways are the same result, but every other combination of global and local gives a different final rotation.