Chapter 5

Combining rotations

Instead of thinking of a rotation as an absolute facing, you can think of it as a change – an offset. In other words, `Quaternion.Euler(0,90,0)` could mean facing east, or it could mean a quarter circle spin from how-ever you're facing 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 the arrow `v1` by `q`. Re-using the star is like how we re-use the `+` to mean push-together in `"cat"+"fish"`. We're not actually multiplying the rotation numbers together. We're running combine rotation equations. But mathematicians actually call it quaternion multiplication.

This is some of the hardest stuff, but if you have the basic idea, you can usually trial and error to get an equation working.

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

vr = y90*vr; // vr spins from right to backwards
```

We're not thinking of `y90` as an eastward facing anymore. Now it's a free-floating clockwise 90-degree y-spin. It's like a card in a game that says "turn right".

Arrow-rotation works with any kind of spin and any arrow. This finds an arrow to the red cube, then spins it 20 degrees on y:

```
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;
```

Imagine the start of the arrow glued to us. The 20 degree spin moves it like a clock hand. The ball will be the same distance from us as the red cube is. If the cube was at 4 o'clock, the ball will be at 5 o'clock. It will be at the same height as the red cube.

Way back in the game board example we started with only the two bottom corners. To get the others we took the arrow along the bottom, pretended it was glued there, and spun it backwards 90 degrees to aim at the top-left corner. We had to use a cheap trick then. Now we can do a real 90 degree spin:

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

  Vector3 acrossArrow = lowerRight-lowerLeft;

  // spin the acrossArrow to get an upSide arrow:
  Quaternion spin90back = Quaternion.Euler(0,-90,0);
  Vector3 upSideArrow = spin90back*acrossArrow;
```

Now `lowerLeft + upSideArrow` gives us the upper-left corner.

A neat variation of that is making a pentagon. A little math shows the outside bend at each corner is 72 degrees. I'll start at the lower-left and go counter-clockwise:

```
    4
 5     3
  1   2
```

I'll assume we have 5 little cubes to drop at each corner. The code will use the same arrow, twisting it to walk along each new edge:

```
Vector3 corner=Vector3.zero; // corner #1
Vector3 arrow = Vector3.right*2; // the arrow from 1 -> 2
Quaternion cornerSpin = Quaternion.Euler(0,-72,0);
c1.position = corner;
corner+=arrow; // 1->2
c2.position = corner;
```

```
// now spin, move and place the rest of the corners:
  arrow = cornerSpin*arrow;
  corner+=arrow; // 2->3
c3.position=corner;
  arrow = cornerSpin*arrow;
  corner+=arrow; // 3->4
c4.position=corner;
  arrow = cornerSpin*arrow;
  corner+=arrow; // 4->5
c5.position=corner;
```

It's a little boring, which is the point. The rotating an arrow trick lets us do a boring turtle-graphics-style walk, using 1 arrow and 1 rotation. Each time we use the rotation on the arrow, it spins another 72 degrees counter-clockwise.

A fun trick is gradually rotating an arrow. It's the same as rotating ourself, except with an extra arrow coming out of us. This spins an arrow from forward, clockwise to backwards, repeating. To see if it worked we'll put a red cube at the tip:

```
public Transform redCube;
float degrees=0;
Vector3 baseArrow = Vector3.forward*2;

void Update() {
  // spin goes from 0 to 180, over and over:
  Quaternion spin = Quaternion.Euler(0, degrees, 0); // y-spin
  degrees+=3; if(degrees>180) degrees=0;

  Vector3 arrow = spin*baseArrow;

  redCube.position = transform.position + arrow;
}
```

`spin*baseArrow` is the fun part. It's an arrow forward, clock-spun 3 degrees, 6 degrees and so on. The tip traces out the right half of a circle, forward to back.

`spin` is really a free-floating rotation "slowly turn 180 degrees right". If we change `baseArrow` to `Vector3.right;` it will spin around the back half. Changing it to `new Vector3(3,0,-1)` would spin over whichever weird half-circle started there.

Some notes on the rules for using these:

- You can't flip the order. `spin*v` rotates `v`, but `v*spin` is an error.

- 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 only needed 1 symbol and we picked `*`.

5.1.1 Converting a rotation into its arrow

In the last chapter `LookRotation` converted an arrow into a rotation. Sometimes it's nice to convert a rotation into an arrow: the (000) rotation is the forwards arrow, and so on.

The trick is to start with the forward arrow. Using it is like adding the rotation to 0. `Quaternion.Euler(0,45,0)` is a NorthEast rotation. Quaternion.Euler(0,45,0)*Vector3.forward is an arrow aimed NorthEast.

For any rotation, `q*Vector3.forward` is an arrow pointed the way `q` would aim.

Suppose we need an arrow aimed forwards and right 30 degrees. Normally we'd just write it as (x,0,z), but I don't know the ratio for 30 degrees. Instead we'll create that rotation and use it to make the arrow:

```
Quaternion qy30 = Quaternion.Euler(0,30,0);
Vector3 v30 = qy30*Vector3.forward;
// v30 is a 30-degree tilted forward arrow
```

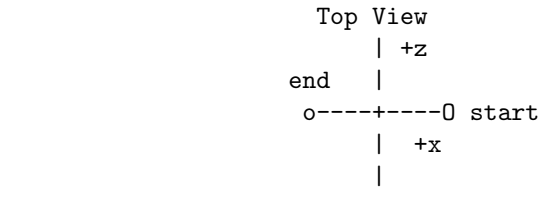
Hopefully this is easy to read. We don't need to imagine the arrow then add that spin. Hopefully we'll see `Vector3.forward` and realize we can think of the spin's facing, and that's the arrow we'll get.

Unity actually uses this trick to turn your rotation into your forwards arrow. `transform.forward` is really `transform.rotation*Vector3.forward`.

5.1.2 More arrow rotation

For an exercise, suppose we want the red cube to start on our right side, then spin left, up and over making a 1/2-circle above us; then repeat.

Plan #1 is this: create a right arrow as the starting spot. As the picture shows, rotating that arrow around z will bring it up and to the left side:



The code for spinning a rightward arrow around z:

```
Vector3 baseArrow = Vector3.right*3;
Quaternion spin = Quaternion.Euler(0, 0, deg0to180);
Vector3 arrow = spin*baseArrow;

redCube.position = transform.position+arrow;
```

That doesn't feel too bad. `baseArrow` is exactly where red starts, and spinning it around z, like a propeller seems obvious enough.

Plan #2 is to make it using pure Euler rotations. Make a rotation that faces right, then leans back up and over to face left. Use `Vector3.forward` to turn it into an arrow:

```
Quaternion spin = Quaternion.Euler(deg0to180, 90, 0);
Vector3 arrow = spin*Vector3.forward;
```

If you can look at `Quaternion.Euler(degs0to180, 90, 0)` and quickly see it as a right arrow, tilting back on x, then this way is better.

Suppose we want to 1/2-circle the red cube from our *personal* right to left. We'll start with the arrow `transform.right*2`. The end is attached to us, spinning around the long line through our body – around our personal z-axis.

This is the cool part: that line is `transform.forward`, and we can make a rotation around it using `AngleAxis`. Here's the code:

```
// our right arrow, spinning around our +z:
Quaternion spin = Quaternion.AngleAxis(degs0to180, transform.forward);
Vector3 arrow = spin*(transform.right*2);

redCube.position = transform.position + arrow; // ball circles R to L
```

That was a slippery one. The trick is to think of `spin` as a free-floating rotation. `AngleAxis(0,transform.forward)` means no spin, which is no change. Changing 0 to 10 means to take whatever arrow and spin it 10 degrees diagonally around our forward arrow, and so on.

As I wrote in the intro, sometimes these are trial and error.

5.1.3 Cones, weirdness

Spinning an arrow around itself does nothing. For example spinning Vector3.up around the y-axis is always Vector3.up again. This code does that – the red cube at the tip won't move:

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

It works, it spins the arrow, But the tip doesn't go anywhere. It doesn't even have a secret spin. The input arrow is (0,1,0) and the output arrow is (0,1,0).

This next code also does nothing. It spins our personal forward arrow around itself. The red cube sits there, 3.5 units in front of us:

```
// forward arrow:
Vector3 baseArrow = transform.forward*3.5f;
// a spin around our forward arrow:
Quaternion spin = Quaternion.AngleAxis(degs0to180, transform.forward);
// this is always the same arrow we started with:
Vector3 arrow = spin*baseArrow;

redCube.position = transform.position + arrow; // never moves
```

Hopefully both of these are totally obvious. If you swing a stick, the tip moves. If you twirl it in place, the tip doesn't go anywhere.

There's nothing wrong with spins that don't move you. It's not an error. If you have lots of changing arrows and ways they spin, it makes complete sense that they sometimes line up for a do-nothing spin.

Spinning funny-angled arrows is like twirling an umbrella and tracking the end of a spoke. If it's all the way open, the spoke-ends make a big circle. As you close it, they make smaller and smaller circles. The spokes haven't gotten any shorter. We're still spinning the entire 2 feet worth of spoke, but the angle gives us tiny circles.

Here's code for an almost-closed umbrella spin. It takes a long almost-up arrow and spins it around the y-axis:

```
Vector3 baseArrow = new Vector3(1,8,0); // 8 up, 1 right
Quaternion spin=Quaternion.Euler(0,degs0to360,0); // spin around y
Vector3 arrow = spin*baseArrow;

redCube.position = transform.position + arrow; // radius 1 circle
```

The red cube is always at y=8, tracing out a small radius 1 circle. It acts like an almost-folded umbrella spoke.

A long arrow at a small angle spinning in a cone is fine. But all the matters is how the tip moves. The area swept out by the long part doesn't really matter.

Math-wise, arrows can be broken into the part along the spin arrow, and the part going straight away from it. That second part is what moves. For example (1,8,0) around y is like climbing up 8 and staying there; and spinning just (1,0,0) to make a small perfect circle.

To sum up: the best spins are when the spin axis and the arrow are at 90 degrees. Other angles are fine, but you may have to work harder to visualize how the tip moves.

5.1.4 Summary

- Use the forward arrow to turn a rotation into its arrow. If `q` is a rotation, `q*Vector3.forward` is the rotation's direction arrow.
- For everything else, don't think of which way a rotation points. Think of how it changes you. Instead of "face right" it would mean "turn right".
- The rule that z won't change your direction is only for rotating 3D models facing forward. For rotations spinning arrows, rotating them on z is perfectly fine and useful.
- Arrows that stick out at 90 degrees from the spin axis are easiest to visualize. But that doesn't have to happen. Non-perpendicular arrows will spin in a cone, which is legal and sometimes useful.
- Any type of rotation works equally well. Quaternion.Euler is good. But `AngleAxis` is often very useful. Especially AngleAxis around someone's personal arrow to spin in their personal space.
- We can now go back and forth from direction arrows to rotations: `q=LookRotation(v)` and `v=q*Vector3.forward` to go back.

5.1.5 Ball cone-shooting example

For a 3D game about fire-hoses, we'd like to shoot little blue balls forward in a small random cone. If we put a wall ahead of us, each ball could hit anywhere within a small circle.

The plan is to start with the forward arrow, which stands for perfect aim. On our aiming circle we'll pick a 0-360 degree direction to miss in. Then we'll pick 0-10 degrees for how far we're off. 0 is the center – we don't miss at all. 10 means we hit all the way at the edge.

That plan should work, but we need another plan to do it:

Version #1 that won't work: make a Vector3 with (x,y,10) where x and y are random -1 to +1. That will give a random mostly forward arrow that can tilt any direction. But the target area will be square. People will notice when we shoot enough, so it's no good.

Version #2 that won't work: Create an Euler rotation that can aim that way. That won't work because we can't make one. We can roll 0-360 on z, but tilting 0-10 on y will only ever go right/left. Or we can compass spin 0-10 on y, but then z-spin will only roll us in place.

That stupid yxz rule is getting us. `Quaternion.Euler(x,y,z)` can't spin y then z like we need.

Version #3: Tilt the forward arrow in 2 steps. First angle it right 0-10 degrees. Second, spin that arrow 0-360 around z. The arrow will trace out a small cone, which is exactly what we want.

The code, in a few parts:

```
Vector3 dirArrow = Vector3.forward;
// small rightward cock:
float missAngle=Random.Range(0.0f, 10.0f);
dirArrow = Quaternion.Euler(0, missAngle, 0) * dirArrow;
```

In our mind `dirAngle` is the actual direction the ball will go. Currently it's cocked 0-10 degrees right. Next we spin it around z. The tip will trace out a small circle:

```
Quaternion zRand360 = Quaternion.Euler(0, 0, Random.Range(0,360)) ;
dirArrow = zRand360 * dirArrow;
```

`dirArrow` is now aimed anywhere within a 10-degree forwards cone. To prove it works, we'll shoot the red cube using it:

```
// ball is 3 units along the aim arrow:
ball.position = transform.position + dirArrow*3;
// speed is in direction of aim arrow:
ball.GetComponent<Rigidbody>().velocity=dirArrow*10;
```

Finally, that always shoots forward. Suppose we want to shoot based on the way we're aiming. That seems like it might be hard, but it's not. We can rotate the arrow by our rotation:

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

Later we'll see a better way to think of this trick. But it's basically taking an almost-forward arrow, spinning it by us, to get an almost-our-forward arrow.

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. Suppose we start rotated `q1` and add a 30 degree z spin. Is that on our current z-axis, or the global z axis? The result will be different.

The good part is, we can pick global or local axis. The bad part is, we do it in a semi-confusing way.

5.2.1 Applying local rotations

When you multiply rotations, the first one is like the start, and the second one spins it more. It uses the local axis of the first. That's the entire rule: "second spin starts with the local axes of the first". It works just great, but it feels really odd at first.

To see it work, let's make 2 simple rotations: a lookAt, and a 360 degree spin around y:

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

// standard y-spin
deg+=2; if(degs>360) degs-=360;
Quaternion ySpin = Quaternion.Euler(0,degs,0);
```

Those two rotations are nothing special. `transform.rotation=redLook;` aims us at red, and `transform.rotation=ySpin;` gives us a clockwise flat compass spin.

But combining them gives something we couldn't do before. It makes us spin in a tilted circle, aimed at red (to see it, red needs to be away from us, and above or below):

```
// titled y-spin aligned with red:
transform.rotation = redLook*ySpin;
```

Think of it as aiming at red, then purposely adding `ySpin` after that. It's only natural it would spin on the tilted y.

We can use 2 combined rotations to solve the fire-hose cone problem. Spin 0-360 on z, then tilt 0-10 degrees to our current right:

```
Quaternion z360 = Quaternion.Euler(0,0,Random.Range(0,360));
float yAmt=Random.Range(0.0f, 10.0f);
Quaternion y0to10=Quaternion.Euler(0,yAmt,0); // small y tilt

Quaternion qAim=z360*y0to10; // <-final rotation
Vector3 aim = qAim*Vector3.forward; // convert it into an arrow
```

Visualize `z360*y0to10` as a cow spinning twice. First `z360` rolls it sideways. It's facing forward, but its right side is spun around to just anywhere. Next it spins 0 to 10 to its *personal* ride side, which for real could be up, left or any direction.

Breaking it into 2 rotation lets us force the order to be z then local y. We had no way to do that before.

In `Quaternion.Euler(-45,90,0)` we know the rule is 90 y first, then -45 on the local x. The word local means the same thing there as it does here. We can break (-45,90,0) into (0,90,0) * (-45,0,0) and it means real y90, local x45:

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

Quaternion y90thenx45 = ySpin90*xSpin45; // same as (-45,90,0)

transform.rotation = y90thenx45; // testing
```

`ySpin90*xSpin45` is just a long way to write `Euler(-45,90.0)`. We wouldn't write it out for real, since writing `Euler(-45,90,0)` is easier in every way.

For a longer example: as we all know, the Mark-II plasma cannon is mounted on a circular base. To aim, the whole base tips straight back. The scary-looking barrel swings side-to-side like the hand on a tilted clock.

In math terms, it uses an xy order – global x, local y. We can't make that with `Euler`. We'll have to use the combine rotations trick:

```
public float xTilt, ySpin; // degrees. Slide-change in Inspector

void Update() {
  Quaternion qTrack = Quaternion.Euler(0,ySpin,0);
  Quaternion qLift = Quaternion.Euler(xTilt,0,0);

  plasmBarrel.rotation = qLift*qTrack; // <-lift, which is x, goes 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 on x, `ySpin` is in a titled arc. If you've ever seen a real Mark-II plasma cannon, you'll recognize the distinctive rotation.

This next one is mostly an excuse to use three in a row. We want the cow to lie on its left side, with its back aimed at the red cube. Here's the plan: aim its face at the red cube; spin it 90 degrees on y – now the left side is aimed there; finally lie down left– now its back is aimed there, with the cow lying on its left side.

In code we make all 3 rotations and combine them in order:

```
Quaternion faceAimRed;
Vector3 toRed = redCube.position - transform.position;
faceAimRot = Quaternion.LookRotation(toRed);
Quaternion spinRight=Quaternion.Euler(0,90,0);
Quaternion fallLeft=Quaternion.Euler(0,0,90);

transform.rotation = faceAimRed*spinRight*fallLeft;
```

I think having three helps see the local rule. First we aim at red. Then spin right based on that. It's important we spin on local y so the left side is aiming exactly at red. Then we flop down left, based on how we're standing now.

Since y beats z, we could have combined them as `Euler(0,90,90)`, but that would have been harder to read. Facing right, then falling feels like it should be 2 steps.

Here's a simpler semi-practical one. We'd like an object to z-spin on it's starting rotation. Without changing it's aim, it will roll in place. We'll save the original rotation, create an increasing z-spin, and add it as local:

```
Quaternion baseFace;
int zDegs=0;

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

void Update() {
  zDegs+=2; if(zDegs>360) zDegs-=360;
  transform.rotation = baseFace*Quaternion.Euler(0,0,zDegs);
}
```

It feels funny since we're recomputing our entire rotation from scratch every time. We take our beginning rotation and apply a 2-degree z-spin. Next frame we do the same thing, but with a 4-degree z-spin.

This trick works to take any arrow and give it a fun roll-in-place.

General local rotation combining notes:

- A rotation by itself isn't local or global. It depends on how you use it. `transform.rotation=q;` is using `q` as global. But `transform.rotation = spin1*q;` uses `q` on `spin1`'s local axes. But then `q*spin1` uses `q` on gobal axis again.

- The "y is always flat" rule only applies for values inside `Euler(x,y,z)`. For example, `q*Quaternion.Euler(20,50,0)` starts in `q`'s local space, then does 50 y first, and 20 x last.

  We often use the combine rotation trick purposely so we can y-spin on some local axis.

- There's no `q1+q2`. There's only 1 rule for combining them, and we picked `*` for it.

- The trick to making the second (and third ...) rotation is to make it as if you were global. `Euler(0,0,90)` lies on your left side. Using it after some other rotation lies on your local left side.

  Part of why we like the combine rotations is how simple that makes things.

5.2.2 Applying global rotations

We can also take a starting rotation and apply to it another as a global. To do that, the starting rotation goes second, and the one to apply as global goes first: `globalSpin*startingFace`.

Let's start with making a global z-roll. z is normally a local spin, so this should look interesting. We'll aim at the red cube and spin on z globally, by putting it in front:

```
public Transform redCube; // look at me
int zDegs=0; // constantly increasing 0-360

void Update() {
  zDegs+=2; if(zDegs>360) zDegs-=360;
  Quaternion zRoll=Quaternion.Euler(0,0,zDegs);

  Vector3 toRed=redCube.position - transform.position;
  Quaternion qToRed=Quaternion.LookRotation(toRed);

  transform.rotation = zDegs*qToRed; // <-spin qToRed on global z
}
```

Since `zDegs*` was first, this will not be a nice local roll on z. The arrow to red will spin around the real z-axis. It will aim it away – far away – from the red cube, probably spinning in a weird little cone. Each full circle will bring it back to looking at red. It's kind of a garbage formula.

You may be thinking "hey wait – you just told me `q1*q2` starts with `q1` and adds `q2` as local. Now you're telling me it's `q2` adding `q1` as global. Which is it?"

Amazingly, it's both. We'll just save that bizarre fact for later.

Going back to horrible arrow spins, suppose I have a 3D game where a cannon aims at a target, but also spins in a circle. You have to tap when it's pointing at the target. It should look like the up/down gear is fine, but the compass gear is running amok. The changes to do that:

```
Quaternion y360=Quaternion.Euler(0, deg,0); // y-spin

Quaternion cannonAim=y360*qToRed; // apply y to aim, as global
```

This gives us the same junky rotation where the arrow to red spins around crezily, but now it makes sense. The blown gasket is merry-go-rounding the aim arrow.

This last one is tricky. We've got a flat spinner that tells us which way the wind blows – North, East and so on. On a signal, everything will tip 90 degrees in the wind direction. Everything tips the same way, since this is just general wind – it's not coming from the spinner.

After some thinking, the wind-blow rotation should be 90 degrees around the spinner's personal x-axis. If the spinner was aimed forward, that flops everything forward – it seems to check out. The math for "90 on spinner's x" is: `Quaternion.AngleAxis(90, spinner.right);`.

Surprisingly, we want that as a global rotation. That's because we're using that axis exactly as it is. Local means we want each object to adjust the wind's x-axis based on their own rotation. That's clearly wrong. With a global wind-flop, the final code is:

```
Quaternion windSpin= Quaternion.AngleAxis(90, spinner.right);
// NOTE: +90 on x is forward&down

foreach(Transform blowMeOver in StuffToBlow)
  // push each item over by 90 in wind direction:
  blowMeOver.rotation = windSpin*blowMeOver.rotation;
```

5.3 Misc rotation combinations

We're back to the bizarre fact that for rotations, `A*B` is either A with B added locally, or B with A added globally. Both things are true. It's one of those crazy math facts.

When you're creating one, it's no problem at all. Suppose you have a plan to start with A, add B on A's locals, then twist it all using global C. That's `C*A*B` (A with B after and C before).

When you're trying to read a compound rotation, it's still not so bad. Think of having two ways to read it, and only one needs to make sense.

Take `lookAtRed*zRoll`. That can be looking at red first, then doing a z-roll on the local axis. Local z-rolls don't change how we aim, so that feels pretty good. Or it's z-roll first: we start aimed North, rolling in place. Then `lookAtRed`, as a global, twists us to look at red, taking the z-roll with it.

Consider the left-lying back-facing cow: `toRed*turnRight*lieOnLeft`. In this we should start with `turnRight*lieOnLeft`. Those take a 000 rotation cow and set it up – it's looking East, lying on its side. That's easy to visualize. Then we apply `toRed` as a global, spinning the whole thing in-place to face the red cube.

That's a useful pattern. Whenever a look-at comes first, think of it as a global. It applies last. Whatever you made with the rest, it gets aimed at red.

From before we know `z360*y0to10` makes a rotation aimed in a small forward cone. One way to read it: 360 spin on z, which does nothing yet. Then the random 0-10 local y-tilt, which could go in any direction of the cone. Or read it as: random 0-10 degree tilt right, followed by a global z-spin which traces out the cone.

The last step of the old cone example was making it face the way we do. That's another example of the "look-at goes first" pattern. `transform.forward*z360*y0to10` is a global `transform.forward` aiming the forward cone.

For fun, we can break `Quaternion.Euler(45,20,180)` into all 3 parts. The official order is always yxz, so we can write it as `y20*x45*z180`. We're allowed to think of that as x first, local z, then global y. Or any order as long as it follows the local/global based on how it was written.

Summary, notes:

- Whenever you create a combined rotation, you have to think about global vs. local. If you have a 90 degree spin on y, is it on the real y, `ySpin*q` or the local y `q*ySpin`?
- The order of a multiply matters! This one almost slipped by. We "know" the order of multiply never matters, but that's only the rule for normal algebra. For rotations, the order matters.
- You can sometimes fix things by flipping the order. Suppose `A*B*C` works wrong, but you're sure A, B, and C are the right rotations. Try moving A to the back, or flipping B and C around.
- You can create a different-order-Euler by doing it one axis at a time, in the order you want. If, for whatever reason, you need a rotation on z, then x, then y, you can force it with `qZ*qX*qY`.
- When you read a rotation, think of it both ways. Read `A*B` as A first, B local; or B first, A global. Usually one way makes more sense.

- z-rolls as the last thing are fun ways to play around without changing the aim direction. LookAt's usually go first, aiming at whatever the rest made.
- To use someone else's local, apply it as a global. If cow1 wants to spin right, it can use (0,90,0) as a local. But if you want to spin around cow1's y-axis, you need `Quaternion.AngleAxis(cow1.up, 90)` as a global.
- The associative property works. In other words, parens don't matter. You can write `A*(B*C)` to make it easier to read. It won't change anything.
- Doing it in parts is the same doing it altogether:

```
Quaternion forwardAimCone = z360*y0to10;
Quaternion meAimCone = transform.rotation*forwardAimCone;
```

- There's no way to make B and C both local on A. The only way is a chain. You have to decide which comes first. ABC means C uses AB's local coords. ACB means B uses AC's local coords.