Chapter 4

Rotations

This section is about setting basic rotations: how to declare a rotation variable, and ways to make and think about them. Later, in another chapter, we'll be able to add them, take fractions and the rest.

For testing, we'll want a good reference object – something where we can easily tell which way it's aiming, and see top/bottom at a glance. If you have a 3D cow or gun pointing along +z, that will be fine. Otherwise I like to make a testing object:

Make a sphere. Then add a cube above it for a hat – child the cube to your sphere and adjust local position to (0,1,0). Then add a forward arrow – child another cube, scale it along z and place it in front (+z):

```
 |  H <- cube hat
+y  O NNNNNNN <- long cube nose
  +z ->
Side view
```

We'll put out rotation code on this, or a cow or gun, to see rotations better.

A decent way to think about a rotation is in two parts: which way it points, and how it rolls. First we aim the nose anywhere, then we roll the hat around.

Or, same idea, point somewhere with your thumb stuck straight out. Then roll your arm so your thumb spins.

That accounts for every possibly way a 3D object can spin.

4.1 Quaternions

Rotations are *not* stored as x, y, z degrees. The Inspector for rotation is a lie. It's actually using math to convert to and from the real way rotations are stored, which we never see.

The real way everyone stores rotations is something called a quaternion. Real programs for robots or 3D space – anything with 3D rotations – have been using them for years. Unity just copied the standard way.

This small example saves our starting rotation and resets it when we press "a" (after you've hand-spun it):

```
Quaternion savedStartFacing; // this is a rotation variable

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

void Update() {
  if(Input.GetKeyDown("a"))
    transform.rotation = savedStartFacing;  // copy back the saved rotation
}
```

Two things are interesting here. `Quaternion` is the type that stands for rotation. We can declare rotation variables like `Quaternion q;`.

The other is that our rotation, `transform.rotation`, is also a quaternion. It's not really x, y, z degrees. It never was. The Inspector is doing lots of work to show us converted values.

`transform.rotation = savedStartFacing;` is copying one quaternion into another. Quaternions are rotations, so it's copying a rotation.

Here's another example that switches my rotation with the red cube's. It uses the standard swap, with a temporary rotation variable:

```
void Start() {
  Quaternion temp = transform.rotation; // save a copy of my rotation
  transform.rotation = redCube.rotation;
  redCube.rotation = temp;
}
```

We can't do much more than this now, since we can't create our own rotations yet.

4.2 Ways to make a rotation

In practice, we like to make rotations in different ways. Setting the x, y, z's for degrees is fine for some things. Other times we want to aim at a green cube, letting the computer figure the angles. Sometimes we want to spin around a diagonal line. Occasionally we want an offset – we're facing A and want the change to face B.

A nice thing about quaternions is they have functions to do all of those. We can set rotations using whatever function seems best for the job, and, later on, mix&match them.

4.2.1 Y, local X, local Z rotations

Using x, y, z degrees is probably the most familiar. It works like aiming an airplane: heading, climb/dive and roll.

Heading is a simple spin on y. It faces us in any compass direction. Next we tilt up or down, along a line drawn through the wings. Together those two things can point us in any direction.

But wait – the line through our wings, based on our y-spin, is our local x axis! It turns out the obvious, best way – the one we naturally use without thinking about it – is global y, then local x. Huh. That's also how Unity reads them.

Finally, we get to roll. It won't change which way we face. It's a spin around the line from tail to nose, which is our local z axis.

All together, x,y,z rotations are global y, local x (based on y,) then finally local z (based on y and x).

What that means is, suppose we want to know which way (20, 90, 284) points. We know z won't matter. 90 y means it's facing east, for sure. Then 20 x means it's tipped a little downward (left-handed coordinates. -20 is up).

When-ever you need to hand-set rotations to face somewhere, spin y to the compass direction, then x for the vertical angle. Then z just for fun.

Euler angles in code

Setting rotations with y, x and z degrees is officially called using Euler angles. In code you create them using the `Quaternion.Euler` function.

The numbers have the same meaning as in the Inspector. This points us east and 20 degrees down:

```
transform.rotation = Quaternion.Euler(20, 90, 0);
```

To make a cow face straight up, we can leave y at 0, then crank x back to -90:

```
public Transform theCow; // link to a cow
  theCow.rotation = Quaternion.Euler(-90, 0, 0);
```

The hardest part is remembering that you can't use just `transform.rotation = new Vector3(-90,0,0);`. Because rotations aren't really x,y,z angles.

We know an x-rotation by itself rolls us forwards. Technically it's making us aim up and down, but it looks like a roll. This would roll a cow forwards, tail over nose, by changing x. It's pretty simple:

```
public float xSpin=0;
```

```
void Update() { // cow roller
  theCow.rotation = Quaternion.Euler(xSpin, 0, 0);
  xSpin+=1;
}
```

Harder, suppose we want the cow facing right, which is y=90, and spinning like it's on a barbecue spit.

x is always a forward roll, tail-over-nose. `Quaternion.Euler(xSpin, 90, 0);` would have us facing right, rolling right.

But not a problem, if we stop thinking about the direction, and think about the type of roll. We want a tilt-roll, and that's z. Here's our sideways barbecue cow:

```
public float zSpin=0;

void Update() { // sideways barbecue-rolling cow
  theCow.rotation = Quaternion.Euler(0, 90, zSpin);
  zSpin+=1;
}
```

Getting these right takes a little practice.

A fun thing, we know y and x aim us. We can make code that uses y and x to actually aim us (using the ASWD keys):

```
public yy=0, xx=0; // heading and up/down
// z will always be 0, since it never affects how we're aimed

void Update() {
  // AD keys spin (y):
  if(Input.GetKey("a")) yy-=1;
  if(Input.GetKey("d")) yy+=1;
  // WS keys raise/lower (x):
  if(Input.GetKey("s")) xx+=1; // s is down, w is up. They look..
  if(Input.GetKey("w")) xx-=1; // ..backwards since +x is down

  transform.rotation = Quaternion.Euler(xx, yy, 0); // aim us
}
```

Simple aiming is best done like this – keep your own copies of the y and x spins. It works pretty well if you can't aim too high or too low.

There's one built-in shortcut for the Euler method. `Quaternion.identity` is short for `Quaternion.Euler(0,0,0)`. This will snap up to facing due-north:

```
transform.rotation = Quaternion.identity; // reset rotation to all 0's
```

4.2.2 Rotate around an arbitrary axis

Sometimes we want to draw just any line through our origin, and spin ourself around that line.

An easy-to-see example, this spins us around a diagonal /-line:

```
public float degrees=0;

void Update() {
  Vector3 spinLine = new Vector3(1,0,1);
  transform.rotation = Quaternion.AngleAxis(degrees, spinLine);
  degrees+=4;
}
```

If you put this on a cube, it will rotate perfectly corner-over-corner diagonally. On a cow, it will do the same thing but it will look a lot stranger (the head will tuck left, then it will be upside-down facing right, then back to normal).

After watching for a while, you should be able to "see" the diagonal line it spins around.

A semi-real example is a y-spin with a small wobble. We'll make a line *almost* straight up, leaning just a tad left, and spin around it:

```
public float degrees=0;

void Update() {
  Vector3 almostUp = new Vector3(-1,10,0); // almost up
  transform.rotation = Quaternion.AngleAxis(degrees, almostUp);
  degrees+=4;
}
```

The arrow counts as a direction – the length doesn't matter. I just thought 10 up and 1 over was easier to read.

Another maybe real use, suppose we're spinning around only y. That's spinning around an up arrow, so we can write it as an AngleAxis. `AngleAxis(degrees, Vector3.up)` is the same as `Quaternion.Euler(0,degrees,0)`, but maybe it looks nicer since it has less numbers.

4.2.3 Look in a direction

This method of setting a rotation doesn't use any degrees at all. We give it a direction arrow and tell it to face that way. The command is `LookRotation`, and the input is one direction arrow.

An simple example, this makes a north-east facing arrow and uses it to aim us that way:

```
Vector3 v = new Vector3(1,0,1); // north east
transform.rotation = Quaternion.LookRotation(v);
```

Obviously this is just 45 degrees. A better use might be an arrow going 3 forward and 1 right, where we can't think of the angle in our head:

```
// look in whatever direction 3 forward and 1 right would be:
Vector3 v = new Vector3(1,0,3);
transform.rotation = Quaternion.LookRotation(v);
```

To aim at a point, like at some other object, we get to re-use our arrow-to-something-else math. This makes us face the red cube:

```
  Vector3 toCube = redCube.position - transform.position;
  transform.rotation = Quaternion.LookRotation(toCube);
```

This is so common – making ourself look at a spot, that there's a shortcut for it named `LookAt`. It takes a point, then computes the line from us to it and all the rest:

```
// I look at the red cube:
transform.LookAt(redCube.position);

// same as:
//   transform.rotation = Quaternion.LookDirection(
//       redCube.position - transform.position);

// the red cube looks at me:
redCube.LookAt(transform.position);
```

All of our old math works with these. If we want to look a little above the red cube, we can do it:

```
Vector3 aboveRedCube = redCube.position + Vector3.up*1.2f;
transform.LookAt(aboveRedCube);
```

A common look-at trick is getting a "flat" spin. Suppose something is on a hill and we want a car to face it by only spinning on y, not tipping back.

We can find the arrow, which slants up, but then flatten it out by setting y to 0:

```
Vector3 toBunny = bunny.position - transform.position;
toBunny.y=0; // now it's a flat arrow
transform.rotation = Quaternion.LookRotation(toBunny);
```

Here's the same trick using LookAt. It makes a fake bunny position, level with us:

```
Vector3 levelBunnyPos = bunny.position;
levelBunnyPos.y=transform.position.y;
transform.LookAt(levelBunnyPos);
```

4.2.4 FromToRotation

This one is a real oddball, which won't be useful until we know how to combine rotations and more math like that. It takes two directions and gives the rotation which would take you from one to the other.

This finds the angle between a northwest arrow and a northeast arrow:

```
NW     NE
 \     /
  \   /
   \ /
90 degrees between them

Vector3 northWest = new Vector3(-2,0,2);
Vector3 northEast = new Vector3(2,0,2);
Quaternion qq = Quaternion.FromToRotation(northWest, northEast);
```

Of course, the result is a 90 degree clockwise rotation on y. But we computed it in a cool way.

It's more interesting when we have arrows to objects. This computes "if you were looking at the red cube, what extra rotation would face you to the blue one":

```
Vector3 toRed = redCube.position - transform.position;
Vector3 toBlue = blueCube.position - transform.position;
Quaternion qq = Quaternion.FromToRotation(toRed, toBlue);
```

The use for these is when we learn how to add rotations, which we can't do yet.

4.2.5 LookRotation's extra roll

`LookRotation` just points us somewhere, and, as we know, that leaves z free to roll. Normally it just leaves it at zero – "head up", But there's an optional input for z-spin.

The rule is pretty funny. Instead of giving a 0-360, you give it a direction and it tries to face your head (your local +y) that way.

Normally use uses `Vector3.up`. This aims at exactly the marker, and also rolls you onto your right side:

```
transform.LookAt(marker.position, Vector3.right);
```

The head direction is like a suggestion. It probably won't be able to face your head exactly that way. It spins on z to whichever gets it the most in that direction.

4.3 Details and math

This last section is unimportant details. A little about how quaternions really work. And then the problems with using xyz rotations. There aren't any useful tricks here But it might make you feel a little better about them.

4.3.1 Real Quaternion values

If you look inside a quaternion, there's an x, y, z and w. You might think those are degrees, but they're not. If you know trig you might think they're radians – still ice cold. They're totally different variables that happened to use x, y and z. If fact, quaternions weren't even invented to be rotations. We used them for other math, then got rid of them when calculus was invented. That's right – people using quaternions thought "calculus is so much easier than these."

If you have an irresistible urge to look at the actual numbers in a quaternion, they're simple 4-float structs. Code to look at them:

```
void Start() {
  q=Quaternion.Euler(0,0,50);
  Debug.Log(q.x +", "+ q.y +", "+ q.z +", "+ q.w);
  // (0.4, 0, 0, 0.9)
}
```

So 50 degrees on z turns into two numbers that make no sense, in slots that make no sense (x and w for z? Huh?) They're not even trig values. You can try it with a few more rotations and they make even less sense.

So, the values in a quaternion are public, but never look at them.

Unity makes them `public` because quaternions are real math, and there are quaternion formulas out there which directly change those numbers. But, really, Euler, LookDirection, AngleAxis and FromToRotation are all you ever need.

4.3.2 dot-`eulerAngles`

If you need your program to look at the yxz degrees of a quaternion, you can get them with the `eulerAngles` function:

```
  Debug.Log( transform.rotation.eulerAngles );
  // shows x, y, z; like in the Inspector
  Debug.Log( transform.rotation.eulerAngles.y );
  // shows just the y-rotation in degrees
```

Remember that the raw x,y,z's, like `transform.rotation.x`, are those nonsense 0.4 quaternion numbers. The `eulerAngle` function does the translating

into degrees.

You can see it's really translating to and from, since the numbers change. This sets using -20 for x, but converting back-and-forth translates it to 340:

```
void Start() {
  // make a rotation then check how it worked:
  Quaternion a1 = Quaternion.Euler(-20,45,0);
  Debug.Log( a1.eulerAngles ); // (340.0, 45.0, 0.0)
}
```

Because the numbers can change, `eulerAngles` is really only useful for testing.

4.3.3 Multiple ways to write an angle

A crazy thing about xyz degrees is there are 2 legitimate ways to write every angle. We can add and subtract 360, like -20 is the same as 340. But forget that. There's one other way involving an up-and-over on x.

For any rotation, you can make it by spinning the opposite way on y, then angling up and over on x, then flipping 180 degrees on z. As an example (0,0,0) is the same rotation as (180,180,180).

Try it with your finger – spin 180 to point to you, tilt up and over 180. That faces you away, palm up. Then roll 180 to be palm-down again.

Another example is (-45,90,0), which is facing right and 45 up. That can also be (-135, -90, 180): face left, but up-and-over 135 degrees, then flip upside-up on z.

It seems like we could make a rule that x has to be between -90 and +90. But that's no good, since we could legitimately get (-100,0,0) by leaning more and more backwards.

That's also (-80, 180, 180), which seems like worse numbers.

This is another reason why you can put a xyz degrees into a quaternion and get different values out.

4.3.4 How the Inspector handles rotation

The Inspector does one more odd thing while it runs your code. It saves the starting Inspector values you entered for rotation, and shows you numbers close to them.

Suppose, before starting, you enter -180 into the y-rotation Inspector slot. That's the same as 180 but using -180 is fine. If you run and your code reads

`transform.rotation.eulerAngles.y` it will see the adjusted value of 180. But the Inspector still shows -180, which is nice.

If you spin one more degree, your code sees 181 and you see -179. Still sort of nice.

The weird part is, suppose your code sets it to 90. The Inspector still thinks you like seeing y-spins close to -180. It will show you -270. It will always adjust the numbers, using mostly +/-360, to show -360 to 0 in that slot.

If you restart with 900 hand-entered there (which is also the same as 180) the numbers it shows you are adjusted

This has no effect on how the program runs, or even what it prints. Only what you see in that side window. If has no effect at all on built games, since they don't have that panel.

4.3.5 Moving with euler xyz problems

One of the confusing things about quaternions is it seems like xy rotations with degrees work just fine, so why bring in these other things. This section is about why xy degrees are terrible.

We usually imagine rotations as a globe. Every spin is the same as a particular spot on it. x and y are latitude and longitude. The two common math things we want involve two points, which was imagine as being near the equator. We want to find the angle between them, and draw that line.

The angle seems easy – compute the x and y difference and use the pythagorean theorem. For example, maybe point A is 60 degrees sideways from B and 20 degrees up. The diagonal is probably about 67. Sure, all three lines in the triangle are curved a little, but not much and maybe it cancels.

Tracing the arrow seems just as simple: proportionally increase both angles. It they hit +60 and +20 at the same time, our arrow should follow the diagonal straight from A to B.

It turns out both of those sort-of work only near the equator. Otherwise there are big problems:

We have to account for shrinking y. If we drag our nice 60,20 triangle further up the globe, 20 x is just as far, but 60 y gets smaller and smaller (just like a real globe – longitudes get closer together, in smaller rings).

The angle method also doesn't account for over-the-top. Suppose you have two points opposite and higher up (75,0) and (75,180). The shortest line is straight over the north pole. That's 30 degrees on x (75+15 to get to the pole, the 15 more to reach 70 on the other side).

Everything is actually a little bit like that. Take points in Canada at the same latitude, just 30 apart on y, but the same x. The shortest path *isn't* a globe spin. It arcs up a little more (imagine there's a second equator, with those

two points on it. That's the shortest line).

Here's an example where the "smoothly change x and y degrees" movement idea is completely wrong. Start 5 degrees in front of the north pole, (85,0) and go to the east side of the equator, (0,90). That should be a line curving down and east, just a little backwards.

But halfway through the two angles will be (42.5, 45). That's way too far forward, and totally not between those two points. The whole thing is a weird forward then back curve.

It turns out that smoothly moving degrees on a globe doesn't make the shortest line. It's always a funny curve, that gets worse and worse as we leave the equator.

All together, xy angle math is good enough near the equator, and turns to junk as you get near the poles. 3D animators work with angles a lot, and know this. For legs, they know not to use a standing globe. Instead, tip it sideways so the normal leg swing counts as being along the equator.

If your game has something that mostly spins, tipping just a little, xy degrees is fine. Angle it (using the child trick) so y is the main way you spin.

If your game is about an actual artillery piece, which for real has trouble using its 2 gears to track things mostly over it, xy degrees are perfect.

But for anything where you can aim anywhere – even just anywhere on the top half of a globe – and want correct math and smooth motion, quaternions are so much better. They don't have poles, or any other spots where spins act funny.

4.3.6 Quaternion setting commands

This section has nothing to do with quaternions or angles. As we know, computers often like to give 2 versions of commands – one that computes the answer, and another that makes you into the answer. We've seen this with `p.normalized` vs. `p.Normalize();`.

The commands from above compute an angle. That's the most useful version. These alternate commands compute and set. For example these two do the same thing:

```
q = Quaternion.AngleAxis(5, Vector3.up);
q.ToAngleAxis(5, Vector3.up); // shortcut
```

Forget about what AngleAxis does. The first one computes a rotation. Then we put it in `q`. The second one computes it straight into `q`. That's the only difference.

But quick review: `AngleAxis(Vector3.up,5)` makes a 5-degree spin on y.

Here are the rest, just so you can say you've seen them:

```
q.SetLookRotation(A);
// same as q=Quaternion.LookRotation(A);

q.SetFromToRotation(A,B);
// same as q=Quaternion.FromToRotation(A,B);

q.eulerAngles = new Vector3(0,90,10);
// same as q=Quaternion.Euler(0, 90, 10);
```

The last one, `q.eulerAngles=`, is confusing. To fit the pattern it should have been `q.SetEuler(0,90,10);`. But it really is a function call. `eulerAngles` is using the C# get/set trick, to be a function in disguise.

And don't let the `Vector3` in it confuse you. It counts as the xyz degrees.

These last three are super-shortcuts. You're allowed to directly assign to your local forward, up and right. It spins you that way.

```
transform.forward = toRedCube;
// same as:
transform.rotation = LookRotation(toRedCube);
```

It makes your forward arrow line up with that arrow. It's main use is if you see LookRotation, and transform.rotation=, and say "whoa, too much!". But all it does is secretly run them for you.

The next two are real oddballs. `transform.right=toRedBall;` gives you a rotation so your right side faces that direction. It really computes LookRotation and adds an extra 90 degree sideways spin.

`transform.up=toRedBall;` is the same. It's like a LookAt using your +y axis.

Most of time I see these used is when someone didn't know +z was forward. They had a cow facing +y, didn't know how to fix it, and used `transform.up=` to aim it.