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.

Our cubes are no good for examining rotations since we can't tell the front from the top and sides. A cow, pointing along +z, is fine (but it can't be tilted to aim that way – its 000-rotation must be +z). Otherwise we can make something. Take one of our cubes and child something unique on top and in front. Maybe a small sphere on top, and forward-aimed cylinder in front:

```
// 3-part testing object, if we don't have a cow model:

 |       o        <- small sphere for a hat
+y       H ===   <-- cylinder as a nose
  +z ->
      Side view
```

Rotations can be imagined in lots of ways. They can be a direction arrow and a roll – like aiming a telescope, then spinning to adjust the viewfinder. Or the old xyz 0-360 method. Or a rotation can be a single diagonal line going through your origin, rotating 0-360 degrees around that one line.

Like vectors, rotations can be offsets. The simplest rotations tell us which way to face. But an offset-style rotation is meant to add to another rotation, or find how far apart two rotations are.

But the various ways to create simple direction-style rotations will be enough to fill this chapter.

4.1 Quaternions

It seems natural to think of rotations as x,y,z, 0-360. That's what the Inspector shows us, and how the 3 circles work using the rotation tool. But those are just tools giving us that view. No one has really stored or used rotations that way for decades. We use a better system called quaternions. The same way `transform.position` is a Vector3, `transform.rotation` is a quaternion.

Quaternions have only one drawback: the numbers in them don't make sense to humans. But that's not a problem since we don't need to look at the numbers. Think of quaternions as a struct with built-in functions doing everything we need.

The simplest way to use a quaternion in a program is saving and restoring a spin. We can declare a quaternion – which is a rotation-holding-variable – and copy our starting rotation into it. Later we can copy it back:

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

// copy our starting rotation into a variable:
void Start() { savedStartFacing = transform.rotation; }

void Update() {
  // spin yourself by hand, then press "a" to restore:
  if(Input.GetKeyDown("a"))
    transform.rotation = savedStartFacing;  // copy it back
}
```

Quaternions looks strange, and it feels weird to copy our rotation into one, but this program is really nothing more than `Vector3 savedPos = transform.position;` and then the reverse.

A similar example, to show using quaternion variables, this switches my rotation with the red cube's. If you remember, a standard swap looks like `temp=a; a=b; b=temp;` In this, `temp` is a quaternion:

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

Still not very exciting, since we don't know how to create or change a rotation yet.

For more using quaternions as normal variables, let's save the rotations of red, green and blue in an array of quaternions. Then pressing 1,2 or 3 snaps us to copy that cube's rotation, using a function:

```
public Transform redCube, blueCube, greenCube; // dragged in links

Quaternion[] Spins; // will be length 3 list of colored cube's rotations

void Start() {
  // basic array creation, with quaternions!:
  Spins = new Quaternion[3]; // array of quaternion variables
  Spins[0] = redCube.rotation; // copy into each array slot
  Spins[1] = blueCube.rotation;
  Spins[2] = greenCube.rotation;
}

void copyRotation(Quaternion r) { // quaternion as an input
  transform.rotation = r;
}

// not very exciting. Keys call the function:
void Update() {
  if(Input.getKeyDown("1") copyRotation(Spins[0]);
  if(Input.getKeyDown("2") copyRotation(Spins[1]);
  if(Input.getKeyDown("3") copyRotation(Spins[2]);
}
```

The point of this is that we can use `Quaternion` like any other variable type. An array of them is fine. `Spins[0]` is the first quaternion in the array. We can pass quaternions to functions. They're regular variables.

4.2 Ways to make a rotation

The most common way to make a rotation is to say what you want to look at, and have the computer do the math. That seems like cheating, but we have a computer – why wouldn't we create that command?

After that, we'll take a look at the old xyz, 0-360 method. We don't have to set rotations that way, but it's a perfectly good option. Then there are a few oddball functions for rotation setting.

4.2.1 No rotation

Unity has one built-in for a preset rotation: `Quaternion.identity` is the 000 rotation, or no rotation. For examples:

```
transform.rotation = Quaternion.identity; // face 000 = forward
redCube.rotation = Quaternion.identity; // same

// save a spin, currently it's "no spin":
Quaternion spin1; spin1 = Quaternion.identity;
```

It's the same idea as `Vector3.zero`. It's a rotation of x=0, y=0, z=0. This is the only preset. For example, there's no Quaternion.up;

It's named `identity` instead of zero since identity is the formal math term everyone uses.

We can think of it two ways. For a tree or rubble or a dirt pile, it means to place it with no extra spin. But for a cow or a flashlight – anything which logically has a facing – it's better to say it aims you North with your head up. But only if the model was made correctly for Unity's coordinates, facing on +z.

That's why I made a big deal about facing +z and possibly fixing it with the parent trick. If your cow faces +x, then `transform.rotation=Quaternion.identity` will face it +x. It will look like the command messed up, but you just have a wrong cow.

4.2.2 Look in a direction

The simplest way of making a rotation is using a direction arrow. Recall that a direction is just any arrow where the length isn't important. Suppose you have (0,10,1), which is an arrow aimed up and a little bit forward. We could turn that into a rotation, assign it to us, and we'll be looking up and a little forward.

The command is `Quaternion.LookRotation`. It converts a direction into a rotation. This code aims us in direction (0,10,1):

```
Vector3 dir = new Vector3(0,10,1); // up, a little fwd
transform.rotation = Quaternion.LookRotation(dir);
```

It almost doesn't seem like we did anything. (0,10,1) is an arrow, and we copied it into our direction, sort of. Except arrows and rotations are different. `LookRotation(dir)` did the math to convert to a quaternion.

This next one is basically the same thing but looking 4 ahead and 1 right:

```
transform.rotation = Quaternion.LookRotation(new Vector3(1,0,4));
```

The cool thing is that there are no angles involved at all. Pretend we're on a board and actually want to look 1 space over for every 4 forward. We don't know the angle and don't need to.

This one is very sneaky. It gives us a rotation of 000:

```
transform.rotation = Quaternion.LookRotation(Vector3.forward);
```

It make us face forward, as advertised. But in Unity, forward, with no up or down, is the starting rotation, which is all 0's.

For fun, here's a hacky way to look in a random forward angle. We'll aim 10 forward and -10 to 10 sideways, which gives a random -45 to 45 degrees:

```
float xAmt = Random.Range(-10.0f, 10.0f);
Vector3 v = new Vector3(xAmt, 0, 10); // a random forward arrow
transform.rotation = Quaternion.LookRotation(v);
```

There's no special reason for using 10. It seemed like a round number, and directions don't care about the length.

Now on to the really useful part: aiming at something. It's easy, since we already know how to get the direction from us to anything else by subtracting points. This aims us at the green cube:

```
  Vector3 toGreen = greenCube.position - transform.position;
  transform.rotation = Quaternion.LookRotation(toGreen);
```

It's so slick. LookRotation takes any arrow, and we have an arrow aimed at green. So, LookRotation faces us to green.. Our aiming arrow happened to be the exact length from us to green. It didn't need to be, but it doesn't hurt. If we had a unit vector, it would work as well.

Aiming the red cube at the green one is the same idea:

```
  Vector3 redToGreen = greenCube.position - redCube.position;
  redCube.rotation = Quaternion.LookRotation(redToGreen);
```

Fun fact, if we flip the subtraction order and get a backwards arrow – green to red, then red will be facing exactly away from green.

Our old vector math tricks work here. Suppose we want to look at a spot a little above the green cube. That's just one more line getting that spot:

```
  Vector3 aimPoint = greenCube.position + Vector3.up*1.5f;
  Vector3 toGreen = aimPoint - transform.position;
  transform.rotation = Quaternion.LookRotation(toGreen);
```

Suppose the green cow can see 6 units, and we want to look at what it sees:

```
  // 6 ahead of the green cube:
  Vector3 aimPoint = greenCube.position + greenCube.forward*6;
  Vector3 toGreen = aimPoint - transform.position;
  transform.rotation = Quaternion.LookRotation(toGreen);
```

Or maybe we want to look at a spot between the red and green cubes. We can use the averaging points trick:

```
  // between 2 cubes:
  Vector3 aimPoint = (greenCube.position + redCube.position)/2;
  Vector3 toMiddle = aimPoint - transform.position;
  transform.rotation = Quaternion.LookRotation(toMiddle);
```

If the 2 cubes are on opposite sides of us, the middle will be in a not very helpful place, but we can't fix that here.

Another common look-at trick is getting a "flat" spin, only on y. For example, we want to y-spin without leaning, to face someone who might be standing on a hill or in a valley. After we get the arrow, set y to 0:

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

I like this one since it feels like "how to rotate, but only on y", which is hard. But it's also "how to aim in a direction, but the direction can't have a change in y", which is easy.

4.2.3 Y, local X, local Z rotations

Euler angles logic

In our code we can create rotations by directly filling in the xyz 0-360 values, just as they would appear in the Inspector. To do that we'll need more details on the exact way they work, and a plan for aiming in a certain direction.

Rotations are made to go in order y,x,z. That seems funny, but it works great. To test, use the slide trick in the Inspector: move the cursor to the left of a number until it turns into a little slide icon, then click and drag left/right. The number will scroll up and down.

How the axes act and combine:

- The y-spin is your compass direction. The y-axis runs up/down, so spinning around y is like standing on the ground, turning to face North/South/East/West. Even if you spin x and z first, y is still a perfectly flat compass spin.
- x-spins are elevation, like setting the angle to fire a cannon. They never change your compass direction – only angle you up/down. Normally we want to set x between -90 and 90. Past those – over the top elevation – aims you in the opposite compass direction that y says, which is legal but confusing.
- z-spins never change the direction you point. They always just "roll" you. I like to imagine a flunky villain aiming a gun, playing around with holding it sideways or upside-down.
- Because of this, it never matters what order you change xyz's. If you side-roll on z, that has no effect on how x and y work. Changing them also keeps that z-roll, on whatever new direction we face. x is the same way – compass spinning y drags any previous x-spin with it.

To face in any direction, set y to the compass heading, and angle x to the elevation. Or go in the other order. And then z is always just for fun. Once the cow looks at what it should, you can roll in onto either side or on its back without changing the aim.

A review of the numbers:

Because Unity thinks +z is forwards, y-spins have 0=north, 90=east, and so on. It goes clockwise:

```
        0      y rotation         -90    x rotation
        ^       top view           ^      side view
        |                          |
270 <--   --> 90                     --> 0
        |                          |
        v                          v
       180                        90
```

Because of the left-hand rule, x-rotations feel backwards. +x tilts downward and -x tilts up. Also because of the left-hand rule, z-spins are counter-clockwise. +z spins to the left.

Some theory: xyz coordinate systems aren't done yet until you say the order. Unity chose to use yxz, which is the best one, but xyz, or zxy, . . . are legal. There's no way to make a system where x,y, and z all just spin you around the real axis. It always has to act like a main axis and a chain of 2 connected gears.

Put another way, it's impossible to say which direction (10,80,30) aims until you know the order. If it's in an xzy coordinate system, it will point in a different direction than Unity would.

That whole system – xyz 0-360 and an order to do them in – is now called Euler Angles. As in "the Inspector shows the Euler angle representation of the quaternion".

Euler angles in code

Because we don't store angles directly, we can't just copy xyz degrees into quaternions. But we almost can. We have a function named `Quaternion.Euler` which translates our Euler angles into the proper quaternion values.

This faces us to the right:

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

It's the same as entering those values into the Inspector. All of these are.

One thing to note is that it's an equals – it snaps our rotation to that exact angle, no matter how we were facing before. We can't add angles yet (it's not `+=`, it's more complicated than that).

This aims us straight up, with out feet facing forwards. Notice how we needed to use -90 to go up. +90 would be facing down:

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

This aims us left, up 45 degrees, and on our back:

```
  transform.rotation = Quaternion.Euler(-45, -90, 180);
```

You need to sort of translate each value: y is clockwise with 0=north, so -90 y is facing West or Left. Negative x still goes up, so -45 x is 1/2-way tilted up. Then z has no affect on direction; 180 z merely rolls us on our back.

Also notice a cool thing about x: it isn't affected by compass facing. That's nice. If we see (-45, yy, zz), we know the cow's height angle is 45-up, no matter what yy is.

You're also allowed to use a Vector3, but it still needs to be an input to Quaternion.Euler. For example this aims South and randomly on our left or right side:

```
Vector3 spinv = Vector3.zero; // stands for a (000) rotation
spinv.y=180; // facing South
spinv.z=90; if(Random.value<0.5f) spinv.z*=-1; // left or right side
// spinv is either (0,180,90) or (0,180,-90)

transform.rotation = Quaternion.Euler(spinv);
```

It's just a shortcut. Quaternion.Euler needs an xyz, which can be 3 numbers, or one Vector3. It can also be a little confusing: a "spin" can now be a real quaternion, a Vector3 holding Eulers, or a Vector3 holding a direction (to be used in a LookRotation).

Using EulerAngles, we can have Update automatically change the angle. This makes a cow do forward summersaults (not backflips, since +x goes down, not up):

```
public float xSpin=0;

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

Here's a trickier one. We want the cow to face right, rolling on its side (we're making a game about roasting a cow). Facing right is 90 degrees on y, and rolling forwards is +x. But this is wrong:

```
   Quaternion.Euler(xSpin, 90, 0);
```

The problem is that x spins in our personal forward. The cow is still doing summersaults. The rules about how x and z travel with y can fool us. We need to think about what rolls in our personal sideways, which is z:

```
// correct right-facing barbecue spinning:
public float zSpin=0;

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

We can play with how x&y together aim us by having pairs of keys control them. A&D spins, W&S aims up/down:

```
Vector3 aimEulers=Vector3.zero; // aiming forward

void Update() {
  if(Input.GetKey("a")) aimEulers.y-=1;
  if(Input.GetKey("d")) aimEulers.y+=1;

  // notice how these go backwards, +1 is down
  if(Input.GetKey("s")) aimEulers.x+=1;
  if(Input.GetKey("w")) aimEulers.x-=1;

  transform.rotation = Quaternion.Euler(aimEulers);
}
```

This is a common trick. We can't just reach into the quaternion and adjust the xyz angles, since they don't exist. So we keep our own copy and send it to our rotation each time.

4.2.4 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); // flat north-East /
  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;
}
```

Again, watching it should eventually help you see the almost-up line we're spinning around.

It might look nicer if the line we spin around is something we can visualize. This code snaps us to in-between the red and green cubes, then spins around that line:

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

void Update() {
  transform.position = (redCube.position + greenCube.position)/2;

  degrees+=3;
  Vector3 redToGreen = redCube.position - greenCube.position;
  transform.rotation = Quaternion.AngleAxis(degrees, redToGreen);
}
```

While running, moving the cubes around will change the line. In a sense, AngleAxis is very simple. It's a single spin around just one axis. If the axis is easy to visualize, AngleAxis looks like a simple spin.

4.2.5 FromToRotation

This command is an improved version of LookRotation. Instead of aiming the front, we can aim any part of us along the arrow. For example, if we're a cow, this aims our feet at the green cube:

```
Vector3 toGreen = greenCube.position - transform.position;
Quaternion feetToGreen =
   Quaternion.FromToRotation(Vector3.down, toGreen);
transform.rotation = feetToGreen;
```

The first input is the part of you to aim, as if you were standing and facing forward. `Vector3.down` always points your down-arrow at the target.

Suppose we want to snub the green cube by looking almost at it. We can do that by aiming an almost front arrow to it:

```
Vector3 almostFront=new Vector3(-1,0,20);
Quaternion aimToGreen =
   Quaternion.FromToRotation(almostFront, toGreen);
```

Since the aiming arrow was slanted a little bit left, and goes straight to green, our head will be facing a little bit to the right.

We rarely need this. Suppose we always want to aim our up arrow somewhere. We'd use the parent trick to spin Up to Forward, then aim our Forward like a normal person.

4.2.6 LookAt

Making us look at a point is so common that Unity has a shortcut command. You give it the point, and it automatically computes the offset, then uses that to aim us.

`transform.LookAt(greenCube.position);` makes us look at the green cube. This command is so useful and common that it's easy to forget that's it's only a shortcut for LookRotation:

```
// this is what LookAt does:
void LookAt(Vector3 pos) { // pos is a point, not an offset
  Vector3 dir = pos - transform.position; // get the offset
  Quaternion r = Quaternion.LookRotation(r);
  transform.rotation = r;
}
```

The drawback is that it automatically aims us. For fun, here's code to fake LookRotation using LookAt. We do a LookAt, read our current rotation as the answer, then reset our rotation to how it was. It's hilariously Rube-Goldburgy, but it works:

```
Quaternion savedOriginalSpin = transform.rotation;

transform.LookAt(greenCube.position);
Quaternion greenQ = transform.rotation; // <- the answer

transform.rotation = savedOriginalSpin; // restore my rotation
```

LookAt works with any object or any point:

```
// the red cube rotates to face me:
redCube.LookAt(transform.position);

// green cube faces red cube:
greenCube.LookAt(redCube.position);

// look at spot in front of red cube:
transform.LookAt(redCube + redCube.forward*6);

// look at the point (10,9,32):
transform.LookAt(new Vector3(10,9,32));
```

The last one is another example of how to tell a point from an offset. (10,9,32) could be an arrow – it would be forward, tilted right and a tad up. But LookAt expects a position, so (10,9,32) counts as a position here.

4.2.7 Directions vs. rotations, LookAt z-spin

A direction arrow is almost a rotation, but not quite. A rotation is made of the direction plus the "free" z-spin. This is the thing where z always goes last and merely rolls you around the direction arrow. A cow aimed at a farmer could be head up, feet up, lying in its side, or so on. Or, again, the goon playing with the cool sideways gun-aiming technique.

A way to see this is that rotations are on 3D models. They have textures, and feet sticking out. We can see them roll. But directions are on arrows. It makes no sense to roll an arrow around itself.

LookRotation turns a direction arrow into a rotation, but since directions don't have the extra z-spin, it cheats. It makes-up a z-spin of zero. In other words, `LookRotation(v)` doesn't give you *the* rotation to face that way. It gives you one possible rotation, out of many.

Later on we'll be able to turn a rotation into a direction. Going from a direction to a rotation then back is safe. It adds a z-rotation of 0, then takes it off. You get back the same arrow you started with. Going from a rotation to a direction then back destroys your z-roll. You'll still be pointed the same way, but your y will always be aimed up.

`LookAt` and `LookDirection` have an option to set your z-spin, but in an odd way. You tell it which way you'd like your +y to point. It's usually thought of as which way your head points. The z-spin will be set as best it can.

These commands will aim you at green, lying on your side with your head pointed right:

```
transform.rotation = LookRotation(toGreen, Vector3.right);
transform.LookAt(greenCube.position, Vector3.right);
```

This shows why we like the "head direction" method. Usually we don't know the z-degrees we want, but know how the top should point. In fact, we couldn't even do this by giving a z-roll. To keep its head facing right, the cow needs to switch sides when aiming forwards vs. backwards.

The thing to remember is this is only a z-roll. The first input is still the real way we aim. The `Vector3.right` means to spin on z to the most right-aiming spin we can get – pick the best out of the 0-360 z-spin.

4.3 `eulerAngles` and round trips

This is a "don't do this" section. It's explaining why, if you want to change your aim with moving xyz degrees, you need to keep your own copy.

The main issue is that when you give the system xyz Euler angles, you won't get the same ones back.

You're allowed to ask for the Euler angles with `eulerAngles`. It gives you a Vector3 with the 3 spins:

```
  transform.rotation = Quaterion.Euler(0,30,120);
  print( transform.rotation.eulerAngles ); // 0,30,120

  transform.rotation = Quaterion.Euler(-90, 0, -20);
  print( transform.rotation.eulerAngles ); // 270, 0, 340
```

The first one gave us the same numbers back, but the second one "fixed" them (for real it recomputed them differently, which feels like it fixed them). Unity prefers 0-360 for y and z. For x it uses only 0-90 and 270-360. The second range is for negative spins; -20 is stored as 340. It cuts out 90-270 since those are over-the-top x-spins.

If you need to change and move xyz Eulers, the best way is keeping your own copies, like the aiming with ASWD keys example. This spins y from 90 down to -90 over and over again:

```
Vector3 vSpin=Vector3.zero; // no rotation -- facing forward

void Update() {
  vSpin.y-=1; // from right to left, then resetting
  if(vSpin.y<-90) vSpin.y=90;

  transform.rotation=Quaternion.Euler(vSpin);
}
```

The system converts negative y's into 270-360, but we don't care since we never look at `transform.eulerAngles`. `vSpin` is like our master copy, and its y is fully controlled by us.

4.4 Details and math

These sections are all comments, background and other things that are nice to see, but aren't really important.

4.4.1 Real Quaternion values

If you look inside a quaternion, there's an x, y, z and w. I already wrote that they aren't degrees. If you know trig you might think they're radians – nope. They're totally different things called x, y and z (and w).

If you have an irresistible urge to look at the actual numbers in a quaternion, it's a simple 4-float struct. This code would show them:

```
public float x,y,z,w; // will be copied from the quaternion

void Update() {
  Quaternion q = transform.rotation;
  x=q.x; y=q.y; z=q.z; w=q.w;
}
```

For 50 on z, (0,0,50) this gives (0.4, 0, 0, 0.9). That means turning on z uses the x and w slots, with numbers that make no sense.

Some more:

rotation	quaternion
0,0,0	0, 0, 0, 1
0,90,0	0, 0.71, 0, 0.71
90,0,0	0.71, 0, 0, 0.71
180,0,0	1, 0, 0, 0
90,90,0	0.5, 0.5, -0.5, 0.5

There doesn't seem to be much of a pattern. Obviously there is. You can find the formulas. But there's nothing useful you can do with those numbers that isn't already in LookRotation and so on.

If they're so unhelpful, why aren't they `private`? That's because quaternions are real math, and actual mathematicians using Unity might want to directly play with x,y,z&w to do something obscure.

4.4.2 Multiple ways to write an angle

A crazy thing about xyz degrees is there are 2 legitimate ways to write every angle. I'm not talking about +/-360 tricks. There's another way involving an up-and-over on x.

Whenever x goes past 90, which is aiming backwards what your y says, you can make the same angle by flipping y by 180 and recomputing x. Here we can rewrite a 135 degree x-rotation as a 45-degree going the other way:

```
A                          A
 \  x=135              x=45 \
  \                          \
   o -> y=0          y=180 <- o


           side view
```

That isn't quite right, since the first way put us on our back, whereas the second keep us feet-down. Flipping z by 180 is the rest of the trick.

The two identical rotations in the picture are (-135,0,0) and (-45,180,180). The first is shorter to read, but the second makes it more obvious that we're actually facing South and are upside down.

Every rotation can be rewritten like that (add 180 to x and z, make x=180-x). Every rotation has a version with x from -90 to 90, and another with it past 90. Fun fact: (0,0,0) is the same as (180,180,180).

In case you were wondering, that's very strange, and we don't like it. But that's the way it is.

The net effect is that moving and hand-checking Euler angles is even more of a giant mess. If all you do is aim mostly forward and keep x and y both 90 or less, things are fine. But with free-movement, anything like `if(q.eulerAngles.y>180)` is doomed to failure.

4.4.3 How the Inspector shows rotations

The Inspector shows a different version of the rotation values. It's not the same as the ones `eulerAngles` computes in the code. Unity saves the starting Inspector values, and keeps what it displays close, using +/-360. It thinks you will like that.

Suppose you start a rotation at (0,0,0) – those are the values entered into the Inspector. While the program runs, the Inspector adjusts everything to between -180 and 180. If a value goes to 181, which is fine inside the program, the Inspector shows it as -179. If you started the x rotation at 90, the range for x would be -90 and 270. When your program sees 271, the Inspector shows -89.

It seems like a huge problem, but hand-checking Euler angles is already such a mess that this doesn't make it much worse. And if you keep your own xyz rotation variable, it will be fine.

4.4.4 Moving with euler xyz problems

We use quaternions because Euler angles have problems. But what are these problems? As we've seen, Euler angles are pretty good for setting a facing. So far that's all we've done, so they seem fine. It turns out that Euler's are terrible at moving between angles.

Basically y&x based rotation movement always works like an army tank aiming its gun. When targets are near the ground, we can track them pretty well. We rotate the turret with y, x is and up/down for hills and valleys. Anything higher up will give us more trouble. It takes less motion for the same degrees on y, making it easer to outrun our turning.

Suppose a flying saucer zooms up directly overhead and we angle x up to 90, tracking it. It can go a little sideways and be safe. Even though the tip of our cannon only has to move a tiny bit sideways, we'll need to spin our turret a full 90 degrees to be able to tilt left.

If you're making that game, you should use Euler Angles for rotation. Use the trick where you keep them in your own Vector3. It will give the weird gear-based gun rotation you want.

The problem happens because most things aren't army tank guns, and look terrible when the work that way. Usually we're looking at the red cow and want to smoothly turn to face the blue one. We don't want the camera to have a little extra curve, or sometimes go really slow with an extra tight funny little spin.

Unlike a 2-gear Euler system, quaternions don't have any good or bad spots. They don't have any problem areas like when a cannon aims mostly up or down. Going from any spin to any other, quaternions always gives a nice straight angle, at the same speed

If you've seen the term Gimbal Lock (the Flying Saucer example has it), quaternions don't have that.

4.4.5 Quaternion setting commands

Our commands all *compute* a quaternion, like:
`q1 = Quaternion.LookRotation(v);`. They have alternate versions where a quaternion sets itself. I've never used them, and they don't do anything new, but seeing the alternate form is a fun review.

Here's the old and new version of LookRotation:

```
Quaternion q;
```

```
q=Quaternion.LookRotation(toGreen); // old way
q.SetLookRotation(toGreen); // new way
```

The second one computes the rotation and puts it directly into `q`. It seems more efficient, but you can't use it in a formula. Not having to write `q=` isn't that much of an advantage.

You could write `transform.rotation.SetLookRotation(toGreen);` since your rotation is a quaternion. But no one ever does. `transform.LookAt(greenCube);` is a better shortcut.

Recall AngleAxis spins you around any line. Spinning around Vector3.forward rolls you sideways. Both of these tip you 5 degrees left:

```
transform.rotation = Quaternion.AngleAxis(5, Vector3.foward); // old
transform.rotation.ToAngleAxis(5, Vector3.forward); // new
```

If our cow has a unicorn horn sticking up at 45 degrees, these will aim it at the green cube:

```
Vector3 hornAngle=new Vector3(0,1,1); // 45 up and forward
Quaternion q;
q = Quaternion.FromToRotation(hornAngle, toGreen); // old
q.SetFromToRotation(hornAngle, toGreen);

transform.rotation = q;
```

The strangest one is the shortcut for setting xyz Euler Angles. It doesn't fit the pattern. It looks like an assignment statement, but it's a disguised function call:

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

Again, quaternions don't actually save the Euler angles. The second command is just running the first one.

There are also 3 shortcuts for FromToRotation – aiming a different part of you somewhere:

```
transform.forward = toRedCube; // same as LookRotation
transform.up = toRedCube; // aim your back that way
transform.right = toRedCube; // aim our right side

// same as:
transform.rotation =
  Quaternion.FromToRotation(Vector3.right, toRedCube);
```

These are more tricks with disguised functions. They actually call From-ToRotation to do the work.

We can use the backwards arrow trick to assign to the other 3:
`transform.up = -toRedCube;` points your feet at the red cube, by pointing your head in the exact opposite direction.