Chapter 4

Rotations

This section is about setting basic rotations: how to declare a rotation variable, and ways to make and think about them. A later chapter is about how to add them, take fractions or spin towards an angle instead of just snapping.

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. Remember Unity thinks +z is forward, and no rotation is looking directly north, flat along the ground.

If you have a 3D cow or gun pointing along +z, that will be fine. I like to make a testing object: make a sphere. Then add a cube above it for a hat – child the cube to that sphere and adjust local position to (0,1,0). Then add a forward arrow – child a another cube, scale it along z and place it in front:

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

A decent way to think about a rotation is which way it points us. In other words, where our local +z faces. That's part one. Part two of a rotation is the local roll on z, which won't change where we're aimed. For my little guy, it would roll the hat around.

Or, same idea, point your arm somewhere with your thumb stuck straight out. Then you can angle your thumb in a circle by rolling your arm.

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":

```
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 shows us the 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, so uses a temp 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 make rotations in different ways. Setting the x, y, z's for degrees is fine for some things. Other times we start out knowing a certain spot and want to aim ourself at it. Sometimes we want to spin around some diagonal line. Other times we want a rotation "offset" – the extra rotation to change from facing A to facing 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

We can set rotations is using x, y, z degrees. It works like aiming an airplane: heading, climb/dive and roll. Or, the same, thing: aiming the gun of an army tank.

First we spin to face a compass direction, then we angle up/down.

Math-wise, this is a rotation on global y, then local x.. That sounds funny, but the part where we angle up/down has to be on our local x, after spinning, or it would be really hard to use.

An example: (-45, 90, 0) says to spin 90 degrees first, spinning from north to east, or 12 o'clock to 3 o'clock. Then tilt up 45 degrees, still facing east (the left-handed coordinate system means -x is up. That can be a pain to remember.)

The last part of rotation is a local z-roll (spinning your thumb.)

It seems funny, but the correct order is global y, local x, local z. Y then x aims you, then a free roll. This is how Unity reads the xyz rotation numbers in the Inspector.

It can help to play with it. Take the long-nose ball with-a-hat and spin it around. Spinning y always gives a perfect compass spin. Spinning z always just rolls the hat around. Spinning x always cranks it up and down without changing the compass direction (unless you completely flip over.)

Even if you do them in a funny order, like x, y, more x, some z, back to y . . . , Unity reads the final results in the order y, local x, local z.

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.

They have the same meaning as in the Inspector, This points you left (-90 y) and a little bit up (-10 x):

```
transform.rotation = Quaternion.Euler(-10, -90, 0);
```

Remember it's not simply setting the x, y, z degrees. It says to make a quaternion, using the Euler method with those angles.

This next example makes the redCube face straight up (no spin means facing forward, then 90 degrees up):

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

This next one keeps us facing right, while gradually rolling forwards (like we're on a barbecue spit):

```
public float zSpin=0;

void Update() {
  transform.rotation = Quaternion.Euler(0, 90, zSpin);
  zSpin+=1; // passed 360, which is fine
}
```

It's using the rule that z is always last, and is on our local axis. No matter how we're facing, the z-angle is always just a roll.

Here's a longer example using the AWSD keys to aim ourself in a small area:

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

void Update() {
  // AWSD keys aim us:
  if(Input.GetKey("a")) yy-=1;
  if(Input.GetKey("d")) yy+=1;
  if(Input.GetKey("s")) xx+=1; // positive is down
  if(Input.GetKey("w")) xx-=1;

  // limit them:
  yy = Mathf.Clamp(yy, -45, 45);
  xx = Mathf.Clamp(xx, -60, 0); // -60 is up

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

If you want to think of a rotation as x,y,z degrees, this is the way to do it. Keep your own variables, like the `xx` and `yy` here, hand move them, and use `Quaternion.Euler` to "apply" them.

It works well enough for a limited area (this only lets us aim mostly forward,) but breaks down if we can aim anywhere.

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 axis

Another way to make a rotation is to pick an arbitrary line for an axis and spin around it. You give the command 1 line, and 1 0-360 degrees.

This gradually spins us around a line running from south-west to north-east:

```
public float degrees=0;

void Update() {
  Vector3 northEastDiag = new Vector3(1,0,1); // a northeast pointing line:
  transform.rotation = Quaternion.AngleAxis(degrees, northEastDiag);
  degrees+=4;
}
```

If you watch a few full rotations you'll spot the diagonal north-east line it's spinning around.

The name might fool you a little. It says `AngleAxis`, but you don't need to give it an official axis. Give it any arrow. It counts as a direction (which means the length doesn't matter,) sticking out of your origin. We spin around it like is was an axis.

This is the same as the previous except the axis line points almost straight up. We get almost a boring y-rotation, but just slightly off:

```
public float degrees=0;

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

It's only a direction, so 10 up and 1 was just an easy way to say "mostly up."

This next example uses our local x-axis to roll something else. It places the redCube in front of us, and rolls it as if we were pushing it:

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

void Update() {
  redCube.position = transform.position + transform.forward*2;
  redCube.rotation = Quaternion.AngleAxis(degrees, transform.right);

  degrees +=4;
}
```

AngleAxis uses the left-hand rule, based on which way you point the arrow. That means if you flip the arrow to go the other way, it flips which way plus and minus go. If that example used `transform.left` it would roll the same way, but towards us instead of away.

This last one uses the arrow from us to something else, because we can. It spin us around the arrow to the red cube:

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

void Update() {
  Vector3 toRed = redCube.position - transform.position;
  transform.rotation = Quaternion.AngleAxis(degrees, roRed);

  degrees +=4;
}
```

The line to the red cube might be long or short, but we rotate around it the same no matter how long it is.

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);
```

The most common use is making look at some other object. We use the arrow pointing from us to them. This keeps us facing the red cube:

```
void Update() {
  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 computes the line, runs LookRotation and sets it as our new rotation, all in one line:

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

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

Remember `LookRotation` takes any `arrow` you supply. But `LookAt` takes a point, which to uses to create the arrow.

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

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

A really fun look-at trick is making a "flat" look-at. We often have the ground will hills. We want to y-spin to face something on the ground, but don't want tip backwards to look up at something on a hill.

a way to do that is to get an arrow to the target, 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 find the angle between a northwest arrow and a northeast arrow. We know in advance it's 90 degrees, but we're computing using between-two-arrows:

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

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

It's more interesting when we have arrows to objects. This figures out if we were looking at the red cube, what extra rotation would we need to be looking at the blue one:

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

The only use for these is when we learn how to add rotations, which we can't do yet. We'll be able to look at the first spot and use `FromToRotation` as an offset, adding it gradually to our first.

4.2.5 LookRotation's extra roll

If you remember, just pointing at something is only part of a rotation – we still get a local z-roll. `LookRotation` points us somewhere.

If you don't add anything else, it leaves 0 at zero – your head is facing up, or as much up as it can be through just spinning.

`LookRotation` has a special rule for the z-roll. You can optionally give it the direction you'd like to try your head to face, as much as possible. It probably won't be able to match it exactly. It just spins on z so your head is facing as much that way as it can.

This aims you, and tries to make your head facing right:

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

No matter what you put for the second input, you'll be aimed exactly where you wanted. It only influences the z-roll.

You almost always want to use `Vector3.up` – head facing up. If you don't write a second input, `LookRotation` uses that.

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."

You may have an irresistible urge to look at the actual numbers in a quaternion. Like Vector3's, they're a simple struct, with just 4 floats named `x`, `y`, `z` and `w`.

We can take a look at the values by setting a rotation with degrees, than looking at those row xyzw values:

```
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 a single 50 degree rotation breaks into two different numbers, both very small, and neither one on the z slot.

You can try it with a few more rotations and they make even less sense.

From our point of view, the actual values in a quaternion are random small numbers totally unrelated to the degrees we entered. But again, that's not a problem since the built-in functions do the work for us.

One funny thing – if they make no sense, why does Unity make them `public` variables? It's because there are a few people who understand quaternions. For example, you could use a copy specialized quaternion function the internet (but Unity has all of the common ones included.)

4.3.2 dot-`eulerAngles`

We can make a quaternion from degrees using `Quaternion.Euler`. We can go back to dgrees using the `eulerAngles` function. That's how the system fills in the Inspector values.

Examples:

```
void Start() {
  // this prints out starting x, y, z rotation:
  Debug.Log( transform.rotation.eulerAngles );

  // 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)
}
```

Notice how the second one changed -20 to 340. That's because it's a round trip. The angles were turned into quaternion `a1`, then translated back. Unity happens to prefer the range 0-360.

Especially because of the way it flips around the values to one's it likes, there's no real use for `eulerAngles` except testing.

4.3.3 Multiple ways to write an angle

Besides being able to add and subtract 360, there's one other way to write the same angle: as 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 spinning an extra 180 degrees on y. 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.

It seems like those up-and-over versions are bad, but sometimes you really rotate x more and more until you go up-and-over past 90.

What this means is every rotation has several ways to write it using x,y,z degrees.

4.3.4 How the Inspector handles rotation

The Inspector does one more odd thing when it shows you the x,y,z numbers. It saves the starting values that you entered, and adjusts the rotation numbers to stay close to them.

It doesn't change the rotation. It uses +/-360 and the up-and-over trick to pick the best numbers out of all the possible ways to write it.

This means your program won't see the same numbers as the Inspector. Suppose you set `transform.rotation = Quaternion.Euler(-20,0,0).` It you print it in the program, you'll see it as 340. What you see in the Inspector depends on the starting value.

If you had 0 in an Inspector slot, the system will try to adjust numbers into -180 to 180. If the "real" value was 190, it would show you -170.

But if you started with 300 degrees in that slot, it would keep the numbers between 120 and 480.

This is suppose to prevent the Inspector values from snapping as much. It's not a real problem since the Inspector isn't part of the final game anyway.

It's just one more reason to to try not to think in degrees, and use the built-in quaternion functions.

4.3.5 Gimbal lock, xyz problems

The biggest problems with using xyz-degree euler rotations is trying to get a smooth spin from one facing to another at a constant speed.

At the equator, they work great. Slowly changing y spins sideways, slowly

changing x spins up/down. But as you angle x upwards, the y-degrees-per-distance gets larger. That slows down the spin-speed.

The worst part is when you're facing anywhere near the north or south pole. If you're aimed mostly up, you can't tilt sideways unless you snap-turn in place on y. That's called Gimbal Lock. It's also really hard to use x and y degrees to smoothly rotate anywhere past the north or south pole.

Animation in 3D models often uses xyz degrees. They solve the gimbal lock problem by limiting where you can rotate, and angling the axis so you mostly rotate around the equator.

Quaternions let us make a smooth rotation from anywhere to anywhere else. They don't have those bad north and south poles.

4.4 Quaternion setting commands

The previous commands compute and return a quaternion. There are alternate commands which assign to an existing quaternion. They do exactly the same thing – they're just alternates.

For example, these are the same:

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

There's nothing special about the second way – I don't think I've ever used it. I just don't want you to be confused if you see someone else using these shortcuts. Here are the rest:

```
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 is kind of confusing – you'd think it would be `q.SetEuler(0,90,10);`.

There are three more super-duper shortcuts: `transform.forward = Vector3.right;` is a shortcut for `transform.rotation = LookRotation(Vector3.right)`. There's no way to set the "up" direction (it always tries to put you head-up.)

You can also align your right side and up with a vector: `transform.right=v1;` and `transform.up=v1;`.

You rarely need those oddballs. Often it's because your model wasn't made facing +z and it's better to use the parent trick to fix that. We can make these later by combining a LookAt and a 90 degree spin.