Chapter 8

Using local space

Sometimes we have a problem where we think "if I was always sitting at 000 facing forward, this would be easy." That's the local space trick.

We've already seen the easy version: solve a problem as if you were facing forward, then twist it to the real angle. Shooting a ball in our forward cone was like that.

The advanced version of the trick is the really useful one. You start with a problem involving you and some other objects. First you bring them into your local space, then solve it using easy math, then bring the answers back into real world space.

8.1 Local Space

Local space is the official name for coordinates using your xyz axis, with you as 000.

So far, we've been using local space informally, for example placing a ball ahead of us using `transform.forward*2`. In our minds, the ball is at (0,0,2) in our local space. Or when you child an object, the coordinates you see are in the parent's local space.

The main trick to using local space is we know the real xyz axes are straight, and our local axes are diagonal lines, but forget about that. Instead, imagine we select and center ourself, and spin the camera to line up our local xyz's. Now, to us, our local coordinates are a perfect, real xyz.

When you're in local space, it's a world where you are always at (0,0,0), pointing exactly forwards. This makes a lot of math very easy.

The reason we don't have to think about the real xyz is we'll use one equation, at the end, to convert back.

Converting local space to global should look familiar. Multiply the local coords by our rotation and add our position:

```
Vector3 localOffset = new Vector3(1,0,2); // sample local-space coords
Vector3 pos = transform.position + transform.rotation*localOffset;
```

Converting a world point into our local space is new. But it's just the opposite: subtract our position and apply the opposite of our rotation:

```
// convert ball.position into our local space:
Vector3 toBall=ball.position-transform.position;
Quaternion qMyInv = Quaternion.Inverse(transform.rotation); // my opposite rotation
Vector3 ballLocal = qMyInv*toBall;
```

A fun question is asking whether (1,0,2) is world space or local space. Like everything else, it depends what you were thinking when you made it. Whenever you have a point, you got it a certain way, or made it for a certain reason. That tells you whether it counts as global or local.

8.2 Local space tricks

A common, obvious use of local space is asking whether a ball is to our left or right. We just have to convert the ball's position into our local space:

```
// standard global to local formula:
qToMyLocal = Quaternion.Inverse(transform.rotation);
Vector3 localBall = qToMyLocal*(ball.position-transform.position);

if(localBall.x>0) print("ball is to my right");
if(localBall.z<0) print("ball is behind me");
```

Notice how "bringing the ball into local space" just means we use temp `localBall` to hold the local space numbers.

Here's a quick check: suppose `localBall` is (1.3, 0, 6.2). That means we can start from us, go 1.3 on our right arrow, 6.2 on our forward arrow, and we're at the ball.

That example only had to bring things to local space – it didn't need to convert anything back to global. Here's an example of the full trick – bring it to local, do the math, bring it back out:

We'd like to suck the ball to our forward/backward axis – if it's 3 ahead of us and 1 up, we want to suck it so it's just 3 ahead. The math for that using real xyz is hard. Here it is using local space:

```
// get local space ball coords:
qToMyLocal = Quaternion.Inverse(transform.rotation);
Vector3 localBall = qToMyLocal*(ball.position-transform.position);
```

```
// Now do local space math. We want to kill the ball's x and y:
Vector3 suckTo = new Vector3(0,0,localBall.z);
localBall = Vector3.MoveTowards(localBall, suckTo, 3*Time.deltaTime);

// now convert back to global and put the ball there:
Vector3 ballPos2=transform.position+transform.rotation*localBall;
ball.position = ballPos2;
```

It looks a little long. But the first is just converting to local space, and the last is converting to global. Once you use this trick a few times, that stuff just goes there and your eyes just brush past it. The middle part is the real work, which is easy now that we're in local space.

This problem is a little made-up. If we wanted to suck the ball directly towards us, we could use normalized `toBall` from a few chapters ago. We wouldn't need local space. But you get oddball stuff like this a lot. Whenever the explanation of a problem involves a local axis, the local space trick might help.

Another use for thinking in local space is not having to use `transform.forward` anymore. When we use that, we're thinking local +1 z, and then immediately translating it into global. If you like local space, we can build a position in local, then turn it into global.

A really simple example is putting something in a random box in front of us:

```
Vector3 toBall; // will be local space
toBall.y=0;
toBall.z=Random.Range(3.0f, 5.0f); // 3-5 in front of us
toBall.x=Random.Range(-2.0f, 2.0f); // a little random left/right

// now convert to global:
ball.position = transform.position + transform.rotation*toBall;
```

We could have built the same thing using `transform.forward` and `transform.right`, but this is a little shorter. It's also nicer if you have complicated placement math and have to adjust x, y and z a lot.

If seems like this would be extra-hard if we want the ball to avoid a dog or something – we'd be test-converting into global a bunch to compare. The usual trick is to bring the dog into local space with us.

Here's one more example of the full (to local, math, back to global) trick:

Suppose we want to keep a rigidbody ball locked onto a diagonal line. The line is marked out by an empty gameObject as one end, with its +z along our line. We can use the local trick: bring to ball to the line's local, fix it (make sure x and y are both 0) and then convert back to global and put the ball there:

```
public Transform theLine; // the start of a line, going in +z
Quaternion qToLineLocal; // saved copy for global to local conversion
public Transform ball;

void Start() {
  // to bring the ball into line's local space
  // computing once, since the line won't move:
  qToLineLocal = Quaternion.Inverse(theLine.rotation);
}

void FixedUpdate() {
  // standard world to local formula:
  Vector3 localBallPos = qToLineLocal*(ball.position-theLine.position);

  // adjust ball to be exactly on the line:
  localBallPos.x=0; localBallPos.y=0;
  if(localBall.z<0) localBall.z=0;

  // standard local to global formula:
  Vector3 ballPos = theLine.position+theLine.rotation*localBallPos;
  ball.GetComponent<Rigidbody>().MovePosition(ballPos);
}
```

There's probably more tweaking to make this particular example work (my ball sometimes bounced extra-hard off the floor.)

8.2.1 Converting rotations, offsets

That math was for points. Sometimes you want to use local space with rotations. It works mostly the same way. Multiply by your rotation to convert local to world; by the inverse to convert world to local:

```
Quaternion qLocalBall;
qLocalBall=Quaternion.Inverse(transform.rotation)*ball.rotation;
// do your math ...

// change back to world:
Quaternion qWorldBall = transform.rotation*qLocalBall;
```

A local rotation of (0,0,0) is pointing along your +z. It means the ball is pointing the same way you are. Local rotation (0,10,0) means to you, the ball seems to be looking a little to the right.

The usual tricks apply, for example you might create a local rotation, then convert to global. Or bring a rotation into local, just to check it.

Converting arrows and offsets to and from local is also done by multiplying by your rotation or the inverse – don't add or subtract your position.

But it works out the same. You probably got the offset by subtracting your position. And you probably intend to add the offset to your position at the end. So this really means to make sure you don't double-add or subtract your position.

8.3 Built-in shortcuts

Unity has a 6 shortcuts for converting to/from local space of a Transform. If you're comfortable with rotation math, learning these is probably more trouble than it's worth. But it's nice to see what they want to provide.

The names are a little funny. I think – I'm not sure – these were the standard names of the commands when Unity borrowed them.

Three of them are different ways to go from local space to world space.

`transform.TransformDirection(v);` is just `transform.rotation*v`. It converts a local arrow to a global arrow.

Notice how it uses the technical term *direction*. That tells you the output is an arrow on global space, and not a point. To use it, you need to add it to something, often yourself.

`transform.TransformVector(v);` is the same, but also multiples by your scale. Which seems like an odd thing to multiply by.

The idea is, suppose you use this to place something touching your surface. If you double your scale, you have to double the offset to still be touching that spot. If that's what you plan to do, this command is perfect.

The math is: `transform.rotation*(transform.lossyScale*v);`. LossyScale is your actual scale, counting your scale and all of your parents'.

The last one is the same except it adds your position at the end. It converts a local into a global point. The command is `transform.TransformPoint(v);`. The math is: `transform.position+transform.rotation*(transform.lossyScale*v);`

How can you tell from the name that that imultiplies by the scale? I don't know – I guess the people who use this just know. Plus most things have scale (1,1,1), so it won't matter anyway.

Notice how all of these take a local position/offset as input. They're the same. A local position is in your local space, which uses you as (0,0,0). An offset from you in your local coordinates is just the long way of writing that.

The other three go from global to local. They convert the input to a point in your local-space. Remember the main use of that is making math easier, then converting back to world space to use it.

`transform.InverseTransformDirection` simply converts a global arrow into your local space. As a review, this uses it to check whether the ball is

to your left or right:

```
Vector3 toBall = ball.position - transform.position; // world
toBall = transform.InverseTransformDirection(toBall); // now local
if(toBall.x>0) Debug.Log("ball is to my right");
```

I suppose you can remember it since the math way to do it uses Inverse:
`toBall = Quaternion.Inverse(transform.rotation)*toBall;`

`InverseTransformVector` is the same, but divides by your scale. It's the opposite of `TransformVector`.

Suppose a basic 1x1 barbecue grill has certain hotspots, and grills scale to any size. This command makes it easier to bring a world offset from a scaled grill into the original 1x1 size, where you can do easy math based on hotspot positions.

Sure, you could just divide by the scale yourself. But if you already know this command the built-in divide is probably pretty nice.

`InverseTransformPoint` assumes the input is a point – not just a global offset. It subtracts you first, then runs inverse transform vector (bring to local, divide by scale.)

And again, these are all just short equations, which you could write yourself. Or maybe seeing the equations helps you use and understand these.