Chapter 8

Using local space

There are many problems that would be easy if we were always at (000) facing forward – for example "is this tree to my left?"

Working in local space is a technique, stolen from mathematics, that helps problems like those. This section is how to do it, and, more important, how we should be thinking.

8.1 Local Space Theory

After playing around with `transform.forward` and `transform.right*2`, and always needing to add `transform.position` you've probably invented the idea of local space for yourself.

Some of the things that become obvious as we do this math:

- Our right, up and forward make a perfect xyz grid. We may as well just call them xyz and write them the normal way, like (2,0,1).
- We won't have any serious equations that move on our x and also the global x. We can say we're working in our local space and then never have to say "our x" and "our y" ever again.

  We can say things like "In our local space, move +2 x, add `v1`, and divide `z` by 2". We don't need to explain how every single movement is using our 3 arrows.
- We're always going to start from our position. It's like our personal (000). There's no reason to write "local space (2,0,1) from us". The "from us" part is included in local space.

Mathematicians simplify it all the way. They think of local space as a fully working xyz coordinate system. It's as if a bunch of people all picked 000's and different xyz's. One is named Global, but it's not any better or different than

the others.

The whole trick is: when you have a problem, first decide which coordinate space makes it easier. If it's some local space, convert into that, do regular xyz math there, then convert back.

8.1.1 Local space and children

As we've seen, when you make something a child, Unity displays the position in the parent's local space. (000) means you're on top of your parent and (2,0,1) means you're 2 right and 1 forward, using your parent's arrows. Those two things are the definition of local space.

As a quick check, we can unchild ourself to see our global coords, then go back in to see local again.

We can use "children get to use parent's local space" for some neat tricks.

Suppose we have a diagonal road and need to place some lampposts and such along it. We want to place one 2.3 left of the road, and the other 2.3 right; and put the next set 4 forward from that, and so on. But the stupid diagonal road makes the math a mess.

Instead, we'll place them using the road's local space. Make a gameObject, place it in the center of the road, facing +z exactly along it. To help get it perfectly straight you could add a cube stretched along z.

Then temporarily make every lamppost a child. That makes them use the road's local space. If two mailboxes have the same x's, they're exactly across from each other. To put a lamppost 4 units further down, add 4 to z.

When we're done, drag them all out and disable the road-empty.

Notice how we never actually wanted them to be children. We invented "road local space" to make the problem easier, put our stuff into it, used it, then put everything back into global.

8.2 Local to global math

We already know how to walk around from us, using our personal xyz's. And it's kind of a pain. Thinking of it as our local space won't let us do anything new, but makes it easier.

Suppose we want to place some things around us, say at `p1` and `p2`. Our new thinking says we'll choose to work in our local space. That means we'll pretend we're 000 and the xyz coming from us is the real xyz, in perfect lines sideways, forwards and up.

Suppose `p1` should be a random amount in front of us: that's `p1=new Vector3(0,0,r);` (pretend we made `r`). Maybe `p2` goes the same amount

behind. We can use `p2=p1; p2.z*=-1;`. To go left, subtract from `x`

That's pretty boring math, which is the point. In this world, we count as being in the most boring possible spot, looking in the most boring possible direction.

We can do harder math – adding arrows, rotating them, percents. It will work the same as it did when we only had one xyz.

The last step is converting to global. It's always the same formula: spin by our rotation, then add our position. It's actually easier than all of those `forward`'s and `right`'s:

```
// convert locals to globals:
Vector3 g1 = transform.position + transform.rotation*p1;
Vector3 g2 = transform.position + transform.rotation*p2;
```

Adding our position is what we've always done. Multiplying by our rotation is new, but we've seen that math before – apply our spin to an arrow.

But we'll think of the whole equation as the formula for converting local space into global.

Another example, not using anything new: we want to put a ball into a random square in front of us. Squares are simpler in local coordinates:

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

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

To go back to the start of this chapter: we're placing the ball exactly as if we were are (000) facing forward. The first 4 lines are very clean. Then we bring it from local to global with the standard formula (which in theory anyone would recognize and understand what we were doing).

8.3 Bringing into local

The next harder problems deal with "where is the tree?" stuff. We want to work in local space, but the points we have are in global. We need to bring them into local.

The equation is the opposite of local-to-global: subtract our position and apply the opposite of our rotation.

This converts the tree into our local coordinates:

```
Vector3 tLocal = Quaternion.Inverse(transform.rotation)*
    (tree.position-transform.position);
```

It's the opposite of the first equation (remember Inverse is the shortest way to get the opposite rotation).

Here's the rest of the tree math. It's very simple, which is the point:

```
if(tLocal.x>0) print("tree is to my right");
else if(tLocal.x<0) print("tree is left of me");
```

For 2 trees we could pre-compute the inverse. This looks a little different, but is still the global-to-local conversion:

```
qi = Quaternion.Inverse(transform.rotation);
Vector3 t1Local = qi*(tree1.position-transform.position);
Vector3 t2Local = qi*(tree2.position-transform.position);

if(t1Local.z>0 || t2Local.z>0)
  print("a tree is ahead of us on the road");
```

With just a little practice, you can spot that equation and see it means we're bringing the trees into local space.

What if we want to know if we're ahead of or behind a truck? Not whether we can see it, but whether it can see us? We can bring ourself into the truck's local space:

```
Vector3 pt = Quaternion.Inverse(truck.rotation)*
    (transform.position-truck.position);
```

Getting that right was a little confusing – use the truck's rotation, and subtracting it's position from us. But it's just plugging 2 things into the equation.

The rest is easy:

```
if(pt.z>0) {
  print("in front of truck");
  // check for narrow strip directly in front of the truck:
  if(pt.z<5 && pt.x>-2 && pt.x<2)
    print("it's going to hit you!!");
}
```

8.4 Round trip local space

The most complicated use of local space is when you need to bring something in, adjust it, then bring it back to global.

Suppose a ball needs to be lined-up exactly on our z-axis. Sometimes it jiggles out-of-line and we need to force it back.

The plan is to bring it into our local space, and change x and y to 0. That snaps it back along our line. Then we bring it back to global.

The code:

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

// snap back to our centerline, which is easy in our local:
bLoc.x=0; bLoc.y=0;

// now convert back to global and put the ball there:
Vector3 b2=transform.position+transform.rotation*bLoc;
ball.position = b2;
```

That really will keep the ball on whatever diagonal line we happen to be facing (even with up and down tilts).

Mentally the first and last part is simply "use local space". The actual work is that little bit in the middle.

8.5 Misc problems

The basics of bringing a point into, out of, or in-and-out of local space will solve most problems like that. But sometimes there are some messy parts to worry about.

8.5.1 Mixing local/global

Mixing local coordinates and globals gives junk results. In theory it's easy to avoid – remember which points are global, which are local, and don't mix them. But it happens.

Supose `windDirection` is a global and we do this:

```
// t1 is tree1 in local space:
Vector3 t1=Quaternion.Inverse(tree1.position-transform.position;

Vector3 leafPos=t1+windDirection*3; // <- mixing local and global
```

Wen we convert back, it's going to be the wrong way. We need to use the local wind direction.

Even more rare is using more than one local space. For a demolition derby game we might use our local space and a truck's to see who's ramming who where.

Our local and truck-local are different and can't be mixed.

8.5.2 Local math from "us"

In our local space, we always count as (000) facing (0,0,1). That's easy to forget when we do math from us.

Suppose we want an arrow from us to the tree. It seems like it should be `t1-transform.position`. But we count as (000). The real arrow is just `t1` minus (000).

It's another example of local space making "relative to us" math easier. It seems weird how `t1` is a position and an arrow, but it depends on how you use it.

The other confusing part is how our forward is always (0,0,1).
Using `transform.forward` is a mistake in your local space.

Like we're been doing, to move something in our forward, add to `z`. That's why we're using local space.

8.5.3 Converting arrows

To bring an arrow into local space, use only your spin. Otherwise it's the same.

Here's a proper use of `windDirection`:

```
Quaternion qi=Quaternion.Inverse(transform.rotation);

Vector3 ballLocal=qi*(ball.position-transform.position);

Vector3 windLocal=qi*windDirection;
```

The middle line is to compare – it's global to local for a point. For the wind, we skip the subtraction and only spin.

An common use is converting a forward arrow. This brings the ball's forward into our local space:

```
Vector3 ballLocalForward = qi*ball.forward;
```

Here's an example going the other way. We figure out a velocity in local space, then bring it into global to use it:

```
Vector3 velLocal;
// set it here, using local coords since it's easier

// now turn it back into global and apply it:
Vector3 vel=transform.rotation*velLocal;
ball.GetComponent<Rigidbody>().velocity=vel;
```

If we never heard of local space we might still do it this way.
`transform.rotation*velLocal` is a standard "spin an arrow by us". But "convert local to global" is easier when you get used to it.

8.5.4 Converting rotations

Rotations use the same rule as arrows – only multiply by the spin:

```
// global to local rotation:
Quaternion ballLocalSpin=
    Quaternion.Inverse(transform.rotation)*ball.rotation;

// local to global rotation:
ball.rotation=transform.rotation*ballLocalSpin;
```

Local spins confuse me. I'd rather use local forward arrows.

But think of it this way: you've got a problem where you want to use local space, and also need to compare your spin with my spin. It's already hard. Bringing the other rotation into local will make it less hard.

8.6 Built-in shortcuts

Unity has 6 shortcuts for converting points and arrows to and from local space. If you know the equations, there's no reason to use these 6.

But we may as well see them:

- `transform.TransformDirection(v);` converts local arrows into global. It's just `transform.rotation*v`.
- `transform.TransformVector(v);` is the same, but also multiplies by your scale. More on that later.
- `transform.TransformPoint(v);` converts a local to global point (spins and also adds your position). But also multiplies by the scale.
- There's no version that only converts local points to global.

Multiplying by your scale is usually no effect, since most scales are (1,1,1).

The reason it's there is because children's local positions scale by the parent. A child with position (1.5, 0, 0) is 3 to your right if your x-scale is 2. As you change your scale, the child moves but it's numbers stay the same (try it).

I'm not sure if that's good or bad. But if you use childrens' positions, multiplying by the parent's scale seems right.

The other three go from global to local. They're the opposites of the three above:

- `transform.InverseTransformDirection` converts global arrows into your local space. It's just your inverse rotation times the arrow.
- `InverseTransformVector` is the same, but divides by your scale (yes, divides – that makes the math work out).

- `InverseTransformPoint` converts a point from global to local, and also divides by your scale.

Mostly you'll see `TransformPoint` and `InverseTransformPoint` used converting to local and back, with the scale left at (1,1,1).

8.7 Summary

The point of local space is being able to say "I wonder if this is easy in local space?"

Imagine how you'd solve it at (000) and no spin. If that way is easy, convert to local space. Usually it's way, way easier – like you were doing diagonal line math, and now you're comparing `x`'s.

If the problem involves rotations or forward arrows, yikes! But you can convert those into local space too, and the result is still probably going to be easier.

Using a few `transform.forward`'s and `right`'s is fine. Technically you're doing all global space math. If you use them a lot and things get messy, consider switching to shorthand, using only (2,0,1)-style numbers. Then convert later.

It so happens distance math is fine without it – for example finding the nearest enemy. Most angle-to math is also fine, even is-this-in-my-view-cone problems.