Chapter 8

Using local space

There are quite a few problems which are only difficult because we could be anywhere, aimed in any direction. If we couldn't rotate, they'd be a lot easier. If we could never move away from 000 they'd be even easier than that.

The Local Space trick says that you can do that. Whenever something is difficult because of your spin, you can pretend you don't have one. While you're at it, pretend you're always at 000. Solve the problem that way, and you can easily turn it into the real answer.

8.0.1 Local space and children

Here's a warm-up using parent/children to place objects. First a quick review:

When you make something a child, Unity starts displaying position using its 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. When we un-child, we don't move anywhere, but the position numbers snap – Unity recalculates your position in global space.

Now onto the example. Suppose we have a diagonal road and want lampposts and mailboxes along the sides. It you've ever tried something like that using the real x&z arrows, it's a mess.

Here's our plan: create an empty gameObject named `roadLocal`. Put it in the middle of the road, rotated to aim along it. Then temporarily make every lamppost and mailbox a child. Adjust them, and un-child when done. That's the whole trick.

It works because of local coords. Since they're children, sliding mailboxes on x&z goes perfectly along the road. If one mailbox is at (3,0,8) the one on the other side can flip x to be at (-3,0,8).

Basically, we figured out a way to make the road count as if it started at 000 and went perfectly along +z. We used the parent/child trick, which made a perfectly slanted local space that allowed us to pretend.

8.1 Local to global math

Suppose we want to pick a random spot in front of us. We want it to be somewhere in a square area with local z from 3 to 5 and local x between -2 and +2. We could solve that with `transform.forward` and so on, but I'm sick of that.

The plan will be to pretend we're at 000, with no spin. We'll determine the spot, and then adjust to account for where we really are:

```
// pick the spot as if we were at 000, no spin:
Vector3 localSpot;
localSpot.y=0;
localSpot.z=Random.Range(3.0f, 5.0f); // 3-5 in front of us
localSpot.x=Random.Range(-2.0f, 2.0f); // +/-2 right and left

// when done, adjust for our real position and spin:
Vector3 spot = transform.position + transform.rotation*localSpot;
```

The first part is boring, which is good. There isn't a single extra term snuck in to account for spin. It's all from chapter 1. That's why this trick is so great. Doing things as if you were at 000 with no spin tends to be pretty easy.

The last line shifts the spot to account for our position and spin. With very little effort, `spot` is now somewhere in a square, aligned with us, in front of us.

That line is the standard conversion from "pretending at 000"-land to the real coordinate system. We'll see the same line every time we use the trick.

In this next example, we'd like to, once again, spin the red cube from our right side, up and over to our left, then snap back and repeat. First we make it as if we were at 000, no spin. That's easy – spin (4,0,0) around the real z:

```
// simple rotation around z:
degs+=2; if(degs>180) degs=180;
Quaternion zSpin = Quaternion.Euler(0,0,degs);
Vector3 redPos = zSpin*(Vector3.right*4);

redPos = transform.position + transform.rotation*redPos;
```

The first part has no weird diagonal lines and no `transform.right`'s. It's not completely simple, but it's not too bad. When done, the standard line converts it. `redPos` is now moving right to left over us, based on our spin.

In this last one, I want the red cube to shoot straight out from me, starting a little bit up and ahead of me. It should go out 4.6 units from where it started, then repeat:

```
float redDist=0; // 0 to 4.6
```

```
void Update() {
  redDist+=0.6f*Time.deltaTime; // it moves at 0.6 per second
  Vector3 redPos=new Vector3(0,1,1.5f); // starting position
  redPos.z+=redDist; // move forward

  redCube.position = transform.position + transform.rotation*redPos;
}
```

Once again, the code to move the cube is downright boring. `redPos.z+=redDist;` moves us forward. Yawn. Pretending we're always at 000 with no spin takes all the fun out of it.

As usual, the fact that we're not really at 000 with no spin is fully taken care of in the last line. It looks exactly like the last line in every other Local Space code.

8.2 Bringing into local

The other type of Local Space problems are when we want to look at other objects. For example "is that tree to my left"? If we didn't have a rotation, that would be easy – just compare x's. If we were also always at 000 it would be even easier – +x is to my right, -x is to my left.

Once again, the Local Space trick says we can do that.

The conversion lines go in front. They get the xyz's of where the object looks, to us. This would find where a tree is:

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

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

That first line looks even odder, but it's also a standard conversion. We'll see it every time we need to examine another object as if we were at 000 with no spin.

The same as the other version, the math is pretty boring. We're at 000, looking forward. If the `treeLocal` is (2,0,-1) then the tree is 2 spaces to our left and 1 space behind us.

If we needed to check several items, maybe 2 trees, we can save some time by pre-computing the inverse. The math will look about the same:

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

```
// check both trees:
if(t1Local.z>0 || t2Local.z>0)
  print("at least one tree is ahead of us");
```

What if we want to know if we're ahead of or behind a truck? That means we care about the truck's rotation, and not ours. We can do the trick as if the truck was at 000 with no rotation:

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

// check my position in truck land:
if(meInTruckCoords.z>0) {
  print("in front of truck");
  if(meInTruckCoords.z>5) print("pretty far ahead");
  else if(meInTruckCoords.x>-2 && meInTruckCoords.x<2)
    print("it's going to hit you!!");
}
```

The math in the conversion line was a little tricky. But, as usual, the rest is super easy. If our z is truck-coordinates is positive, we're in front of it. If our truck-coordinates x is between -2 and 2, we're in the path.

8.3 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. We'll use both equations, in the normal place they go.

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, change x and y to 0. Then compute it back to global.

The code:

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

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

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

The first and last part are the standard conversions. The middle is the work, and it's so easy it doesn't seem like we did anything. If you're at 000 facing along +z, how do you force something to your forward arrow the shortest way? Get rid of its x and y.

8.4 Local Space Theory

As you may have guessed "pretending you're at 000 with no rotation" isn't an official math thing.

For real, we're choosing to move around using our local axes, the same as always. When we're inside the trick and write (0,0,3), we're actually starting at our position and following our personal `transform.forward` for 3 units.

The secret of Local Space is how it feels like pretending we're at 000 with no rotation. When we decide to work in our local space, we mentally fade out the real xyz's. We do math as if our grid is the only one. We purposely ignore the real xyz's until we're all done. The secret is that in every way that matters, our Local Space is a perfect world with us at 000 and no spin.

Suppose we use `Quaternion.Inverse*(tree1.position-transform.position)` and get (-1,0,0). For real, us and the tree are probably way off somewhere, but close together. If we look at how we're rotated, the tree is exactly 1 unit along our left arrow. But once you know all of that, you can forget it. "The tree is at (-1,0,0) in our local space" is a quick and useful way to say all of that.

Later on, `pos.x+=2` really means to walk 2 spaces on our `transform.right`. The Local Space trick says to relax and stop trying to do two things at once. Walk around on Local as if it's a regular grid. Let x+2 just be x+2. Convert later.

The pair of equations – bringing trees in, and moving answers out – are technically local-to-global conversions.
`Quaternion.Inverse*(tree1.position-transform.position)` converts global to local. `transform.position+transform.rotation*A` converts local to global.

8.5 Misc

8.5.1 Converting arrows

We usually bring points into and out of local space. That's what the regular formulas do. But sometimes we need to do that with arrows. The formulas as a little different.

As an example, suppose the real world `windDirection` is (1,0,0). The wind is blowing East. If we need to use it in our local space, it will probably be in a different direction. Likewise, if we compute a local wind direction of (1,0,5), that's in our right and forwards arrows. We'll need to convert to global.

This would bring the wind into local space. It's the usual equation, without subtracting a position:

```
Quaternion qi=Quaternion.Inverse(transform.rotation);
Vector3 windLocal=qi*windDirection;
```

A stranger version for that, suppose we need a ball's forward arrow, in our local space. It's the same equation:

```
Vector3 ballLocalForward =
  Quaternion.Inverse(transform.rotation)*ball.forward;
```

Going the other way, suppose we calculate a velocity in our local space. Multiplying it by our rotation converts it to global:

```
Vector3 localVelocity = ...;

// convert to global:
Vector3 vel=transform.rotation*localVelocity;
ball.GetComponent<Rigidbody>().velocity=vel;
```

Notice it's the same equation as for points, but without adding a position.

8.5.2 Converting rotations

It's not as common, but sometimes we need someone else's rotation, in our global space. To visualize this, suppose we're facing right and a tree is facing forward. To us, the tree is facing left.

The equations are the same as for arrows:

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

// local to global rotation:
tree.rotation=transform.rotation*treeLocalSpin;
```

Since rotations and forward arrows are almost the same thing, I generally use only forward arrows.

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, but it's interesting to see the names.

These go from local to global:

- `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. But also multiplies by the scale.
- There's no version that only converts local points to global.

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.

Multiplying and dividing by the scale is funny. Most things have a scale of (1,1,1), so it won't matter. If you changed the scale, those commands will give wrong results.

But Unity adjusts children's local positions based on the parent's scale. If a child's x is 3 and the parent has x-scale 0.5, the the child is really only at x=1.5. The special commands using scale are meant for child objects, I think.

8.7 Summary, notes

If you need to do a few simple movements, `pos+transform.right*2` is just fine. But suppose you need to do more than that. Compare:

```
// working in global:
Vector3 pos=transform.position;
if( ... ) pos+=transform.forward*3;
if( ... ) pos+=transform.right*1.6f+transform.forward;

// using the local trick:
Vector3 pos=Vector3.zero;
if( ... ) pos.z+=3;
if( ... ) { pos.x+=1.6f; pos.z+=1; }
pos = transform.position + transform.rotation*pos;
```

Thinking in local space, even if we didn't have to, has 1 extra line at the end, but makes the rest of it easier to read.

It seems funny that `transform.forward` is in global space. When we use it we're thinking in local space – moving 1 along our z. But then we immediately turn it into the global coordinates. In local space, "1 ahead of me" is simply (0,0,1). We'd convert it to global later.

The trick is only for when whatever it is we're doing depends on our rotation. Suppose we want to find the nearest object. We don't need this trick since our rotation doesn't affect distance. But say we want the nearest where sideways and backwards count as more than forwards (we hate turning and backing up). Now we want Local Space.

The form is always:

```
o compute local coords of anything we need

o do math in local coords

o convert anything we need to use "for real" into global coords
```

Everything is a variation of this. Sometimes we're merely checking something. We convert it in, do our checks in Local, and there's nothing to convert back. Other times we have nothing to bring in – we compute a position in Local and bring it out.

If we need to "fix" a position, we convert it in, adjust it in local, then convert it back. But we might bring in several objects, use them to compute some other spot, and convert only that back for use.

Deep down, the trick is as simple as it sounds "can I solve this if I was at 000 with no spin, then account for my spin later?"