Chapter 2

Local and Global coordinates

Occasionally it can be helpful to pretend an object has it's own personal coordinate system: (000) glued to the object, xyz spinning with it. We can imagine that grid overlaid on the real coordinates.

Since we have two xyz's, we'll say the real one is *global*, and the one on us is our *local* coordinate system. Obviously there's only one global, but everyone has their own personal local coords.

It turns out that if we know one, we can figure out the other. Global to Local and back. This is great. It means we can place an object using our personal xyz, if that's easier. Then convert into the real xyz. With any luck, we can make the system do that for us.

What often happens is you have a rotated object, often rotating. It could be facing any direction at any time. And the math for "go to my right" and "behind me" and so on is hard – piles of trig hard.

"Try it in local coordinates" can be a magic spell. It says to work out the problem as if you were always facing forward, and also pretend you're always at 000. Our trig problem often turns into grade-school math. Solve that and you're done. Really. The last step is to tell the computer you were using local space. It will account for it, like magic.

2.1 Local/global translate tools

Before math and coding, let's just play with local axis in the Unity3D editor. Even though they aren't real, the tools in Unity let you use them.

Get in Scene view, select any rotated cow and pick the Translate tool (on top: it's the Crossed arrows, between the Hand the the Circle arrows). Red,

green and blue arrows should come out of it, aligned with the real x, y and z axes

A few buttons to the right, you should see a button that says Global. It's a toggle. Tapping it flips between Local&Global. If the object is rotated, you should see the xyz arrows snap to a new spot, then back.

On the Local setting, spinning the object spins your local arrows. As you might guess, +z is always out your front, +x is always your right side. If you turn upside-down, your local +y goes down (which is what up-side-down means).

If you drag a rotated box using the xyz local axes, you'll see it move in a nice grid. It feels like a regular xyz system. Technically, it's all fake. When you move on just 1 local axis, the Inspector shows x, y, and z all changing at once. It's re-calculating the global coords for you. The local axes aren't "real-real". But we can use it and it works great. It's real enough.

Pretty much any 3D program has options for Local/Global. Many users who aren't so hot on math have an excellent understanding of Local and Global axes. It just takes practice.

A note: the colors are standard. Everyone lines up rgb and xyz – the red arrow is always x, green is y and blue is z.

2.2 Using your local axis in code

Since they're so useful, Unity provides your local-axis arrows in code: `transform.right`, `transform.forward` and `transform.up`.

The way I remember is that `transform` means me, and holds my position and spin. So `transform.right` is my right. `Vector3.right` is global right. It never changes, but transform.right has to be calculated based on the spin.

We can use them like regular offsets. This puts the red cube 3 units to our right:

```
public Transform redCube;

void Update() {
  redCube.position = transform.position + transform.right*3;
}
```

Since this uses *our* right, spinning us will move the red cube, keeping it always 3 units right of us.

As normal, we can combine these offsets. This puts the red cube 2 in front and 4 left of us. Again, spinning shows that it's our front and left, based on our facing:

```
redCube.position = transform.position +
```

```
    transform.forward*2 + transform.right*-4;
```

Unity didn't give us transform.left, since by now everyone knows left is negative right.

We can improve our old movement code by using these new arrows. This line in Update moves us the way we're facing:

```
transform.position += transform.forward * 0.01f;
```

You can check that it always uses the our current forwards by running it and spinning a little. Notice how I was too lazy to figure out speed-per-second and then divide by 60. Instead I just guessed 0.01 for slow, but not too slow.

The rest of these are examples of how everyone has their own personal local coordinates. Our script can look at other peoples', and use them like regular offsets.

This code moves the red and green cubes along their forwards (face them in various directions to check):

```
redCube.position += redCube.forward * 0.01f;
greenCube.position += greenCube.forward * 0.01f;
```

As you might guess `transform.forward` is my forward, `redCube.forward` is her forward, and so on. You can ask any Transform, such as redCube, for any of their local axes.

This next example is basically a mistake, but legal. We're trying to move ourself forward, but accidentally used the red cube's forward, not ours:

```
transform.position += redCube.forward * 0.01f;
```

It's pretty weird. We'll move in a funny direction. Spinning the red cube, which isn't moving, will change the direction we move. it's a little like a remote-control steering wheel.

Here's another garbage example, where we position the green cube based on the red and blue forward arrows:

```
greenCube.position = transform.position + redCube.forward*3 +
  blueCube.forward*4;
```

If both are pointing in the same direction, green will be about 3+4 units away. If facing in opposite directions, they mostly cancel out. It's like the green cube is on an invisible 2-part robot arm, controlled remotely by the red and blue cubes' spin.

Here's one where it makes sense to use other people's arrows. Each cube is 2 units in front of the previous one, forming a chain:

```
redCube.position = transform.position + transform.forward*2;
blueCube.position = redCube.position + redCube.forward*2;
greenCube.position = blueCube.position + blueCube.forward*2;
```

The math in all three lines uses the same cube's position and forward. "2 in front of me" makes sense. The chain is fun – if we rotate a cube, everything after it tracks the spin. That looks pretty cool if we do it while running.

Now back to movement code. We can adjust the old code where we shot out a green cube over and over. The movement line can now be our personal forward. The rest of the code is the same. As with all the others, it looks best if the object is rotated, to show how it always goes forward:

```
float pct=0; // from 0 to 1
public Transform greenCube;

void Update() {
  // the new line:
  Vector3 moveLine = transform.forward*8; // 8, since why not

  greenCube.position = transform.position + moveLine*pct;
  pct+=0.01f;
  if(pct>1) pct=0;
}
```

It starts from our origin. For a cow, that's down by the feet, which isn't so nice. We can bring it higher up, maybe out the mouth, by adding an extra transform.up offset:

```
void Update() {
  Vector3 moveLine = transform.forward*8;

  greenCube.position = transform.position +
      transform.up*2 + moveLine*pct; // <- new, adding up*2

  pct+=0.01f;
  if(pct>1) pct=0;
}
```

It's looking a little complicated, but all we're doing is adding 2 offsets. We always go 2 in our up, then go a variable amount in our forward. The zooming camera did the same thing:

```
                  transform.forward*??
                 ----------------green
 transform.up*2 |
               COW
                o
```

It can feel strange not having `transform.forward` come straight out of us. But it's just an arrow representing our personal +z. Moving our up, then our forward feels natural enough.

2.3 Other Unity commands and local coords

This uses the built-in `Translate` command to move slowly in our forward direction. The inputs are x, y, and z. But see if you notice the strange part:

```
void Update() {
  transform.Translate(0, 0 , 0.01f);
}
```

The new/strange/nice part is that we didn't have to do anything funny to get our local z. Translate was written for when you want to move along your arrows. You give it the amounts on them, and it does the rest. Technically, Translate expects the inputs to be in your Local Space.

Suppose we want to move forward and a little up, at a speed of z=0.01 and y=0.002. Written both ways, the Translate one looks nicer:

```
void Update() {
  // old way:
  transform.position += transform.forward*0.01 + transform.up*0.002f;

  // the version with Translate has far less nonsense:
  transform.Translate(0, 0.002f, 0.01f);
}
```

People like to think in local coordinates, and expect to have built-in functions using them. Translate is a useful, simple example of that.

But now we have one more thing to worry about: xyz's can be in global or local. Local is so useful that it's worth being a little confusing. And in most commands it's obvious which they want.

On to the next command. Anything with a rigidbody and a collider will normally fall and bounce around. Unity also has two commands to shove, using global or local coordinates. `Relative` isn't a technical term, but it clearly means local:

```
rb.AddForce(0,0,4); // real +z, like a plunger
rb.AddRelativeForce(0,0,4); // local +z, like a thruster
```

The interesting thing is how (0,0,4) can mean Local or Global, depending on how you use it.

Similar side-by-side commands are common when either way might be useful. First decide whether the motion should be described using your personal

arrows, or the real xyz. That tells you which command to use, then work out the numbers.

Back to Translate. The short version which we saw is automatically in Local. It has 2 longer versions where you can select Local/Global using the 4th input:

```
transform.Translate(0,0,1, Space.Self); // local
transform.Translate(0,0,1, Space.World); // global
```

Self and World words are more informal terms. Some game designers prefer them to Local and Global.

There's no special reason why AddForce is 2 different commands, while Translate is one with 2 options. But it doesn't really matter. The main thing is knowing there can be a local and global version. If we want to give the directions using our personal arrows, we can. We just have to find the command that wants Local Space.

2.4 Childing, `localPosition`

Suppose we have two cubes side-by-side, off somewhere. One might be at (3,0,6) and the one next to it at (4,0,6). If we go to the list of names and drag the second cube into the first, it becomes a child. It's locked to the first cube, tracking it. Nothing special so far.

But another interesting thing happens. The numbers for the second cube's position snap to (1,0,0). It doesn't move, and it's not at (1,0,0), But the Inspector shows those numbers. If we move and spin the first parent cube, the child cube moves to track it, but always displays position (1,0,0).

Moving the second cube does another strange thing. Suppose we pull it further right so it says (2,0,0). Now it tracks the parent based on that new position. The position numbers are still locked, now to (2,0,0):

```
      c2 child, at parent's local (2,0,0)
     /
\   /
 c1 parent facing north-west
```

The rule is: objects with parents use the parent's local space. "Locked 2 spaces right of our parent" is just another way to say "(2,0,0) in our parent's local space". The Inspector is showing our local position in our parent's space, and allowing us to adjust it. The only very minor problem is how the label misleadingly still says Position.

A fun local-space parent trick is snapping one object to another. First make yourself a child of the thing you want to go to. Then enter 000 for your position.

That puts it 000 away from the parent. Neat, right? Finish up by unparenting.

Another fun child trick is sliding yourself based on another object. Say you want to slide yourself along a very tilted wall. Just make yourself a child of it. Now sliding your Inspector xyz's uses the wall's arrows (to slide a number, click just to the left when the cursor turns to a slider-symbol, then slide). x and y slide you perfectly along the surface of the wall, no matter how the wall's been rotated.

You can use the child-trick to play a fun game: get a cow and a ball, neither a child of the other, and spin the cow. The game is to place the ball exactly in front of the cow, 3 units away. Once you think you have it, check your work by making the ball a child of the cow.

The numbers you see will instantly snap to local. The closer you are to (0,0,3), the better you're lined up. If x is almost 0, you got the left/right almost perfect.

Of course, you could get it perfect by making yourself a child, typing in (0,0,3) and un-childing. But playing it like a game is educational.

In code things are different, a little nicer. `transform.position` is always your real-world position. Even children can use this as normal. If you're a child, `transform.localPosition` is where you are in your parent's local space.

Here's a crazy-looking script to run on a child. Just drag yourself into anything and tilt it. This code moves you along your parent's forward:

```
void Update() {
  transform.localPosition += Vector3.forward * 0.01f;
}
```

The key is that `localPosition` is in the parent's local coords. Adding to z is the same as adding to Inspector z, which is local in the parent. Vector3.forward "feels" global, but it's just (0,0,1). What it does depends on where we put it.

It's another case where we can choose Local or Global based on the command we choose:

```
  // [position] is real xyz:
  transform.position += Vector3.forward * 0.01f;

  // [localPosition] is parent's local xyz:
  transform.localPosition += Vector3.forward * 0.01f;
```

This next one is a little silly, showing more localPosition use. Assume the green cube and us share a parent. This puts the green cube where we are, except mirrored on our parent's x-axis:

```
// greenCube needs to have the same parent as us:
public Transform greenCube;
```

```
void Update() {
  Vector3 localPos = transform.localPosition;
  localPos.x *= -1;
  greenCube.localPosition = localPos; // our pos, with x flipped
}
```

To test, we can slide ourself around and see green mirror us – the same spot on the cow, but on the other side.

We can redo the "fire a cube out of the cow's mouth" example using localPosition (the cube has to be a child of the cow). Instead of a percent, this will use meters going from 1 to 8. But otherwise it's the same:

```
public Transform greenCube; // assume this is our child
float zDist=1; // in units. goes up to 8

void Update() {
  // put green 2 above and z forward:
  greenCube.localPosition = new Vector3(0, 2, zDist);

  // move zDist from 1 to 8:
  zDist+=0.02f;
  if(zDist>8) zDist=1;
}
```

`localPosition = new Vector3(0, 2, zDist);` is the fun part. It's a clear, short way to say "2 up and z forward on my parent".

2.5 Looking at the numbers

This section is very optional. It's for if you aren't happy unless you see the numbers and can check them out.

This simple program shows values for your 3 local arrows:

```
// Inspector copies of your local axes:
public Vector3 right, up, fwd;

void Update() {
  right = transform.right;
  up = transform.up;
  fwd = transform.forward;
}
```

If we have no spin, right will be (1,0,0), and so on – our local axes start out lined up perfectly with the real ones. I didn't say this before, but they're also length 1, just like the real ones.

If we spin 90 degrees clockwise on y, right will be (0,0,-1). That's still normal – our right is the real backwards. But tilting 45 degrees gives some strange numbers. `fwd` will be (0.707, 0, 0.707).

That's correct. It's length 1 going diagonal. It turns out 0.707 is the cosine of 45 degrees, and is also 1/2 of the square root of 2, and the pythagorean theorem says the whole thing is length 1.

Spinning 30 degrees makes `fwd` be (0.5, 0, 0.86). That's also length 1, and also the sin and cosine of 30 degrees.

If you tilt on y and x, you'll get even stranger values for x and y and z. But the math says they check out.

Finally, this is one more thing that might be bugging you: the manual says `transform.forward` is in *world* space. What?? We use it for local space positioning, so how can it be world space?

It's about which arrows the numbers are meant for. (0.707, 0, 0.707) is how to make a diagonal arrow, using the real x, y, and z. `transform.forward*3` is like a 2-step process: we think "3 forward local", which is (0,0,3); then convert using the arrow.

2.6 Errors

Mixing local and world axes in the same math usually gives junk, for example: `transform.right*3 + Vector3.right*2`. That's 3 to out personal right, then 2 on the real right. Depending on our spin, they could mostly cancel out, add, or anything in-between. It's not illegal, but it's almost never useful. It's usually a sign you made a mistake.

`Vector3.up` vs. `transform.up` can be an exception. Suppose you want a label above someone and a little behind. `transform.up*3 - transform.forward`. seems correct. It works fine when they spin. But if they tilt or lean, then `transform.up` tilts and pulls the label too much. The label looks better if we go straight up from the feet: `Vector3.up*3 - transform.forward`.

The opposite problem can happen. You place a hat using `Vector3.up*1.4f`. That works perfectly at first, since they rarely bend over. But when they do, the hat is floating above their feet.

The most confusing errors are double-conversions. If something expects local coordinates, you can't use `transform.forward` and its friends. This is legal but gives wrong movement:

```
transform.Translate(transform.forward);
```

`transform.forward` converts into the values we want. But Translate doesn't know that. It assumes they're unconverted locals and does it again. It's a little bit like converting feet to inches twice in a row.

Hopefully this also feels redundant. We use commands taking Local coords so we don't have to use any other tricks. We like how `Translate(0,0,1)` means our forwards.

Double-conversions can be hard to spot during testing, since the wrongness of the result can be a lot or a little, or none if you aren't rotated.

Another double-convert error which is pretty much the same: we're setting our child's localPosition, which wants Local coords:

```
// red is our child
red.localPosition = transform.forward*2; // a bad double-convert
```

To go 2 forward we should have used just (0,0,2). localPosition will handle the rest.

The same error with AddForce:

```
rb.AddRelativeForce(transform.forward*6);
```

We chose to use RelativeForce so we could use (0,0,6). Adding the transform.foward messed that up. Either of these would work correctly. Obviously the second one is nicer:

```
rb.AddForce(transform.forward*6);
rb.AddRelativeForce(new Vector3(0,0,6));
```

2.7 Longer space-fighter example

There's nothing new here, but it's fun to see a few things used at once.

As we all know, X-wing fighters fire a blast from each wing tip, angled inward a little so that they meet after 50 meters:

```
   |= laser . . .
   |             . . . .
>OOOC                     X bam!
   |             . . . .
   |= laser . . .       transform.forward*10 - transform.right*2
```

Pretend we have some tiny rigidbody balls as prefabs, ready to be used as laser bolts. As a quick review, anything with a rigidbody moves by itself. We set the `velocity` once, at the start, in units/second.

This code handles placing the balls at our wingtips:

```
public Transform ballPrefab;

void Update() {
  // the space key fires:
  if(Input.GetKeyDown(KeyCode.Space)) {
    // wing tips are 5 sideways and 2 ahead of us:
    Vector3 toRightWing = transform.right*5 + transform.forward*2;
    Vector3 toLeftWing = transform.right*-5 + transform.forward*2;

    // spawn and place them:
    Transform bRight = Instantiate(ballPrefab);
    Transform bLeft = Instantiate(ballPrefab);
    bRight.position = transform.position + toRightWing;
    bLeft.position = transform.position + toLeftWing;
```

The only tricky part, for me anyway, was the wingtip math. On my first try I flipped the entire left equation, so the ball was 5 left and 2 *backwards*.

Next we set the ball's speeds using the `velocity` built-in. They should fly forward and a little inward:

```
    // set the speed of the 2 balls we just made:

    // right ball moves fast forward and a little bit left:
    bRight.GetComponent<Rigidbody>().velocity =
      transform.forward*10 + transform.right*-2;

    // left ball has a small drift going right:
    bLeft.GetComponent<Rigidbody>().velocity =
      transform.forward*10 + transform.right*2;
  }
}
```

To test this, it looks nicer if you uncheck the balls' UseGravity box. With gravity turned on, they'll probably fall below the camera before meeting in the middle.

2.8 Intro to local space theory

You may have noticed how `transform.forward*2 + transform.right` are basically hacks. They work great, but they aren't they way we like to think about local space.

The best way to do this stuff is to think completely in local, at first. If we want to be 2 ahead and 1 right, we write (1,0,2) and remember it stands for local. Then we either find something that wants local, or convert at the end.

For example, if we want a child at (1,0,2) from us, `c1.localPosition = new Vector3(1,0,2);` is great. It says exactly what we were thinking, without being very long. `c1.position = transform.position + transform.forward*2 + transform.right` will work, but it's basically hiding "local (1,0,2)" from us.

Likewise `rb.AddRelativeForce(new Vector3(0,1,10));` is the nicest way to say that we want a local push, of this xyz local amount.

So far, there's no good shortcut for placing a non-child using our local coords. We'll get one later, but just for fun we can make it now.

We've seen the equation over-and-over: to be (1,0,2) from us, start where we are, and multiply out direction arrows. Here's a function doing that:

```
Vector3 toLocal(Transform tt, float x, float y, float z) {

  Vector3 pos = tt.position; // start where we are:

  // move x,y,z along our local arrows:
  pos += tt.right*x
  pos += tt.up*y;
  pos += tt.forward*z;

  return pos;
}
```

Using it to place the green cube (1,0,2) from us:

```
  greenCube.position = toLocal(transform, 1,0,2);
```

It almost seems like cheating, but that's what functions are for. We can see it's taking transform.right and transform.forward*2, and adding it to our position.

It's also a complete sum-up of the entire chapter. "(1,0,2) from us" is a fine way to think, the computer can do it, and it's even easy to read (once you know about the local trick).