Chapter 2

Local and Global coordinates

Local coordinates are a geometry trick for handling math from someone's point of view.

A common example is wanting to know if a tree is to our left. All we need to do is put it on our personal xyz grid and check whether x is negative.

We don't have a personal xyz grid, but it's easy to pretend we do. It's known as our local coordinates. In them, 000 is always where we are, +z is always ahead of us, and so on. Standard math can translate back-and-forth to the real xyz. Lots of things that are hard to do using the real xyz are easy using our personal grid. We translate back when done.

2.1 Local/global translate tools

The first way most people see local axes are the drag arrows in edit mode.

To see them, get in Scene view, select any object and pick the Translate tool (on top: it's the Crossed arrows, between the Hand the the Circle arrows). The red, green and blue arrows show the x, y and z axes (the colors are always in that order – red is always x, and so on).

The next group of buttons, to the right, should say Pivot and Global. Global means to use the real x,y,z. Clicking it toggles to Local, then back to Global.

If you spin the object, then toggle, you'll see the difference in the arrows. Global is always the same – the real arrows. Local depends on your spin: blue (+z) always comes out of your front, and red (+x) is always to your right. If you tip yourself, green (+y) is your personal up and not the real one.

But these are just a handy way to move – lots of people like to slide things the way they're facing. For real, the system only stores the one real set of x, y,

z's. You can watch the Inspector numbers to check this.

This Local/Global feature is actually standard on any 3D program.

2.2 Using your local axis in code

In code, Unity precomputes the 3 local-axis arrows for everything. For you, they're `transform.right`, `transform.forward` and `transform.up`.

To compare, remember `Vector3.right` is the global x. It's always a boring (1,0,0). `transform.right` is your personal x.

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;
}
```

Those local arrows are length 1, so `*3` scales it to length 3 like normal. Also like normal, we can combine them. This puts the red cube 2 in front and 4 left:

```
redCube.position=transform.position + transform.forward*2
  +transform.right*-4;
```

Using `right*-4` for left is the normal way. It feels like setting x=-4. And by now everyone knows the flipping an offset trick.

This makes us move the way we're facing. Nothing special, but fun:

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

Every object has it's own, which you look up using their transforms. This moves the red and green cubes along their forwards:

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

You're allowed to mix&match, but there's rarely a good reason. This makes us move in the direction the red cube is facing:

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

We won't jump to the red cube. It's only acting like a remote control, telling us which direction we go. Which is just silly.

This positions the green cube from us, based on the blue and red cubes' facings. It's legal, but the final position is nonsense:

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

Those two vectors probably aren't even at right angles, and they have no relation to us or the green cube. We're just following two random arrows.

But here's a fun one where it makes sense to use different arrows. A chain of cubes where each is in front of the other:

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

The pattern "me + my forward*aNumber" makes sense. If we want to spin the cubes to make a zig-zag, that's our business.

An example where we move something along a line won't show anything new, but it's fun. This slides the green cube alongside of us:

```
public Transform greenCube;

float dist=-3; // will go from -3 to 5

void Update() {
  Vector3 toGreen = transform.right + transform.forward*dist;
  greenCube.position = transform.position + toGreen;

  // moves dist from -3 to 5:
  dist+=0.02f;
  if(dist>5) dist=-3;
}
```

If you like the percent-based version, we can also do it that way. Nothing new, but it's fun:

```
float pct=0; // from 0 to 1

void Update() {
  Vector3 moveLine = transform.forward*8; // the entire line
  Vector3 toStart = transform.right + transform.forward*-3;

  greenCube.position = transform.position + toStart + moveLine*pct;

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

It's another one where we have one normal offset, then another scaled.

2.3 Intro to local space theory

The official way to think of your local space is by using normal (x,y,z) vector3's. Two spaces right and one forward is (2,0,1) and we just know it's local space.

When we're done, we translate into real space. Here's a fun function to do that:

```
Vector3 myLocalToReal(Translate tt, Vector3 localOffset) {
  Vector3 pos = tt.position;

  // combine our three arrows, scaled by what they said:
  pos += tt.right*localOffset.x;
  pos += tt.up*localOffset.y;
  pos += tt.forward*localOffset.z;

  return pos;
}
```

It does the same math we were doing before. The difference is how easy it makes it to use just (2,0,1) for local:

```
  Vector3 v = new Vector3(2,0,1); // counts as local
  // put it (2,0,1) in my local:
  greenCube.position = myLocalToReal(transform, v);
```

In general, we like to be able to "think" in local space, using as many vector3's for local x,y,z as we need. `v.x+=1;` moves one more space to our right, and so on.

When we're completely finished, one final standard step converts to real.

2.4 Other Unity commands and local coords

Unity's built-in Translate command uses local coordinates. This moves us forward (our forward):

```
transform.Translate(0,0,1);
```

`transform.Translate(-0.1f, 0, 1);` is forward and drifts a little left.

It's easier than the `transform.forward` method, but more limited – we can't use it to place items.

When you're a rigidbody, Unity gives a command for either way:

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

They thought `Relative` sounded better than local. The Translate command has more synonyms for local/global, in an optional second parameter:

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

Sometimes the real xyz grid is called World Space, and the one based on us is our Local Space.

To sum up: you'll often seen two versions of a coordinate-using command. The exact words may change, but one is global, the other local.

And if you know vector math and offsets, it's often easier to use that.

2.5 Childing, `localPosition`

When an object has a parent, Unity displays it's position in the parent's local space. You may have noticed this: take a Cube just 1 unit in front of another, and drag to make it a child. It's Position numbers change to (0,0,1).

It's a very funny semi-glitch. They could add an extra slot for things with parents, or at least relabel it. But everyone (who uses 3D software) knows the rule about it now being local.

A fun way to use this: make one thing a child of another and enter 000 for Position. It snaps directly to the object – 0 units away from it, in all directions.

In code there's an extra variable for that, named `localPosition`. This moves us forward, the way our parent is facing:

```
  // pretend we are a child of something
  transform.localPosition += Vector3.forward;
```

It's sneaky, at first. Imagine it runs for a while and gets to plain old (0,0,5). The system knows that's an offset, from your parent, the way it's facing. If your parent faces diagonal, you move diagonal.

Here's the "slide a ball from -3 to 5" example, written if the red cube is a child. All of the local position math is gone, since the system does it:

```
public Transform redCube; // assume this is a child of us
float zz=-3; // moves from -3 to 5

void Update() {
  redCube.localPosition = new Vector3(1, 0, zz);
          ^^^^^^^^^^^^^ this makes it work

  // same code as before, moving zz
  zz+=0.02f;
  if(zz>5) zz=-3;
}
```

You're suppose to see `localPosition` and immediately understand x=1 means one to our left, and `zz` means our forwards.

Overall, the idea is that you made it a child for a reason. Expressing children's positions in the parent's local space quickly becomes natural, and the easiest way to get most things done.

2.6 Looking at the numbers

This section is very optional, if you aren't happy unless you see numbers.

Suppose we're tiled 45 degrees on y:

- Our forward will be (0.707, 0, 0.707). You might recognize those as the sin and cos of 45 degrees.
- `transform.up` will be just (0,1,0). Since we didn't tip or lean, our up is still the real up. Nothing wrong with that.
- `transform.forward + transform.right` makes an upside-down V. It ends on the x-axis. But that's obviously correct: for us tilted 45 degrees, the real x shoots right and forward.

This simple program shows values for your local arrows. It uses the trick of copying into Inspector variables every frame:

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

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

With no rotation, you'll see they're the same as the real ones. Spinning 180 degrees flips some signs (forward will be (0,0,-1)). 90 degrees moves them around: forward becomes +x.

If you know sin/cos of 30 degrees by heart, spinning 30 degrees will make (0.5, 0, 0.86).

The manual says `transform.forward` is in *world* space. What?? That's really saying it's ready to use. We think (0,0,2) in local space, then we write `transform.forward*2` to get the world space translation.

2.7 Errors

Mixing local and world axes in the same math usually gives junk, for example: `transform.right*3 + Vector3.right*2`. It's not illegal – 3 to my right, then +2 on x. But it's almost never useful - usually a sign you made a mistake.

It's easy to use `Vector3.up` instead of `transform.up` and not notice the problem. If you never tilt or lean, they're always the same. Sometimes I write `Vector3.up` to purposely get funny results if I tip, since I should never tip.

If something expects local coordinates, you can't use `transform.forward` and friends. For example:

```
red.localPosition = new Vector3(0,0,1); // 1 forward
red.localPosition = transform.forward; // yikes
```

The entire point of using localPosition is to let the system convert for us. `transform.forward` is us converting by hand. It turns (0,0,1) into different numbers, then the system converts them again.

The end result is a weird double-rotation.

The same problem with AddForce. These both push us forward. In the first one, we account for our rotation. In the second, it does:

```
// two same ways to push us on our forward:
rb.AddForce(transform.forward*6);
rb.AddRelativeForce(new Vector3(0,0,6));
```

But `rb.AddRelativeForce(transform.forward);` double-converts.

These become easier with practice.

2.8 Bonus 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.

Pretend we have some tiny rigidbody balls as prefabs, ready to be used as shots. This positions them at the wing tips:

```
public Transform ballPrefab;

void Update() {
  // space 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;
```

Nothing new here, math-wise. I almost used negative rightWing for the left, except that would put it -5 x and *behind* us by 2.

The speeds are both 10 forward, then -2x and +2x to angle them inward. The math part is still nothing new:

```
    // the rest of firing 2 wing-blasts:
    bRight.GetComponent<Rigidbody>().velocity =
      transform.forward*10 + transform.right*-2;

    bLeft.GetComponent<Rigidbody>().velocity =
      transform.forward*10 + transform.right*2;
  }
}
```

Only the +2 and -2's are flipped. Really, it's just two boring lines coming from our wingtips. Except setting them as the `velocity` makes the balls automatically shoot that way.

One note: rigidbody's start with gravity turned on. That's fine, but for a spacelike feel, uncheck that box.