Chapter 2

Local and Global axes

2.1 Local axes

When we have a rotated object, it might be handy to imagine it has a personal x, y and z. The official name for these are its local axes.

When we want to talk about the real x,y,z, we call those the global or world axes.

To see both, select some object with a spin, go to top of the Scene window, select the Translate tool (shows red/green/blue arrows) and change the Global/Local toggle at the top of Scene. You'll see them change from the real x, y, z's to your personal ones. If you spin yourself, Global mode keeps your arrows locked, and Local mode has them spin with you.

This feature is standard on any 3D program, not just games.

The purpose of local axes is just to be more arrows we can use. If we want an arrow pointing the way we're looking – that's our local +z axis. Having them can complicate things – whenever you use an axis you have to think global or local. But they're worth it.

2.2 Using your local axis in code

`Vector3.right` is the *world* x-axis. Each object has its own personal local axes, since everything can be rotated a different way. You have to ask an object for them with `transform.right`, `transform.up` and `transform.forward`.

Those give us the exact red, green and blue arrows we see in local translate mode. We didn't think of this much before, but they're always length 1. As we spin around, the exact xyz numbers making those arrows will change, but the length works out to 1.

Fun fact: 3D systems match RGB colors with xyz: the x arrow is always red, y is always green, z is blue.

The opposite directions don't have a built-in – there's no `transform.left`, but that's not a problem since we know scalars. `-transform.right` flips it to go left.

We can use these as regular arrows. Here's a silly one putting a red cube 3 spaces to our right. If you spin yourself, the dog will spin with you:

```
public Transform redCube;

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

The magic part is how the system automatically recomputes `transform.right` as we spin, so it's always our local +x. Besides that, our code is just the same old point+offset math.

We can combine these arrows, the same as before. This puts the red cube 2 in front and 1 up (spinning with us):

```
void Update() {
  redCube.position=transform.position + transform.forward*2 + transform.up;
}
```

A fun trick with local axes is making us move the way we're facing. This adds part of our forward arrow to ourself each update :

```
void Update() {
  Vector3 mv = transform.forward*Time.deltaTime;
  transform.position += mv;
}
```

If we press Play and spin ourself a little in the Inspector, we'll see this really moves the way we're facing. Again, the magic is how the computer looks at our spin and auto-computes `transform.forward`.

The same local-axis math can be used to shoot a ball. This uses standard placement with an offset to start the ball in front of us. The new part is using our forward vector to set the ball's speed (it also assumes you know how to use Unity prefabs, and rigidbodies):

```
public Transform ballPrefab;

void Update() {
  if(Input.GetKeyDown(KeyCode.Space)) {
    Transform bb = Instantiate(ballPrefab);
```

```
    Vector3 toBall = transform.up*1.0f + transform.forward*2;
    bb.position = transform.position + toBall;

    Vector3 ballMove = transform.forward*5; // medium-slow the way we're facing
    bb.GetComponent<Rigidbody>().velocity = ballMove;
  }
}
```

The velocity of an object is just an offset – it's how it moves during 1 second. We set it once and the system figures out how much to move each frame so it adds up. `transform.forward*5` shoots the ball at 5 units per second, going the way we're facing.

One note: rigidbody's start with gravity turned on. You'll see the ball fly straight, then quickly curve away as it falls. If you want to see it fly exactly straight, find the UseGravity checkbox in the ball prefab's rigidbody and uncheck it.

We can use the usual vector math to play with the velocity arrow. This fires two balls, from our left and right, and angles them both a little inward, so they crash in front of us:

```
void Update() {
  if(Input.GetKeyDown(KeyCode.Space)) {
    Transform bLeft = Instantiate(ballPrefab);
    Transform bRight = Instantiate(ballPrefab);

    bLeft.position = transform.position + transform.left*2;
    bRight.position = transform.position + transform.right*2;

    Vector3 fmv=transform.forward*5; // forward speed, but not done yet

    // angle them both a little inward:
    bLeft.GetComponent<Rigidbody>().velocity = fmv+transform.right;
    bRight.GetComponent<Rigidbody>().velocity = fmv+transform.left;
  }
}
```

I like this example since it shows how the rules are always the same. Once we know `velocity` is just a vector offset, we can use regular offset rules. If the left ball has velocity 5 forward and 1 right, that's a regular slightly diagonal arrow, same as any other.

Local axis work just as well with moving numbers. Adding `-transform.up+transform.forward*-3` to our position gives a point a little below and 3 units behind us. Changing the -3 to -2, -1 ... moves it in closer. As usual, we can make it move by turning that into a variable. I'll have it move from -3 to 5:

```
public Transform redCube;

float dist=-3; // will go from -3 to 5 (then wrap around)

void Update() {
  Vector3 toRed = transform.up*-1 + transform.forward*dist; // <- dist changes
  redCube.position = transform.position + toRed;

  // moves dist from -3 to 5:
  dist+=2*Time.deltaTime;
  if(dist>5) dist=-3;
}
```

As we spin, the track the red cube takes will follow us.

The previous version of this had one long arrow (from the fixed cube to us) and used 0 to 1 percents to move along it. For this it seemed easier to use the length 1 `transform.forward` arrow and let the numbers stand for actual distances. Either way is fine, depending.

We can use other people's local axes. `redCube.forward` is the direction the red cube is facing. This is a simple example using our forward and its forward. We move in ours, and move the red cube using its:

```
public Transform redCube; // spin so it has a different facing than we do

void Update() {
  transform.position += transform.forward*1*Time.deltaTime;

  redCube.position += redCube.forward*0.8*Time.deltaTime;
}
```

This next one uses everyone's local axes to make a chain of cubes. red goes in front of us, green goes in front of red (based on how *red* is facing) and blue goes in front of green. Twisting any cube will shift the ones in front of it:

```
void Update() {
  redCube.position = transform.position + transform.forward*2;
  greenCube.position = redCube.position + redCube.forward*2;
  blueCube.position = greenCube.position + greenCube.forward*2;
}
```

There's no rule that `greenCube.forward` has to start at `greenCube.position`, but it's an example of the "where would that arrow make sense" idea. You could add green's forward arrow to the red cube, but it's hard to think of a good reason.

2.3 Things that assume local coords

Using local axes is so handy, some commands assume you want to use them. For example, the built-in translate command assumes you're thinking of local coordinates:

```
transform.Translate(0,0,1); // local +1z
// same as:
transform.position += transform.forward;

transform.Translate(2,3,0); // +2 to your right, +3 local up
// same as:
transform.position += transform.right*2 + transform.up*3;
```

I used this shortcut in the previous chapter in the "spin and move the way you're facing" testing movement code.

The key thing is that `Translate` is just a different word for changing your position. If you're mostly a 3D modeler, and are comfortable using the translate tool to move things, and know the Local setting is often the most useful, then this command makes perfect sense.

But if you know vector math there's no reason to ever use this. Vector math, `=` and `+=` is more flexible.

Pushing a rigidbody (increasing the speed) has an option for global or local. Suppose you have a 3D model with a facing, like a cow, and `rb` is its rigidbidy. These two commands shove it in different directions:

```
rb.AddForce(0,0,4); // cow starts flying north (real +z)
rb.AddRelativeForce(0,0,4); // cows starts flying direction its facing
```

The first one is a shortcut for adding 4 to our z-speed. Math in world coords is simple like that. Written out, it's the same as:
`rb.velocity += Vector3.forward*4;`
The second one isn't much worse. It looks up the cow's local forward:
`rb.velocity += cow.forward*4;`

The word `Relative`, in `AddRelativeForce` is just an informal way of saying Local. When I see `AddForce`, I wonder whether it's local or global, since either would make sense. But then seeing the other one, I figure "since `AddRelative` is local, just `AddForce` must be global."

Besides *relative*, Unity has some more words that are just informal ways of saying local and global. The translate command, from before, has local and global options:

```
transform.Translate(0,0,1, Space.Self); // local - moves my forward
transform.Translate(0,0,1, Space.World); // global - +1 on real z
```

It sounds nice to say "move me 0,0,1 in my local space," but it's just another way of saying to use your local axes. Likewise "world space" sounds nice enough, but it means "using the global axes."

And there's still no reason to use these if you know vector math.

To sum up: you'll see various commands with two options, probably using odd words. They're always just a choice between global or your local.

2.3.1 Looking at the numbers

This section is for if you're feeling a few things aren't quite right, and want some more explanation. If that's not true, you can safely skip ahead.

"For real," Unity uses global axes for everything. Positions are only ever stored using the real x, y, z, and movement can only be done using global coordinates. `transform.forward` doesn't really tell the system to use your local forward. It does, but the way it does is by recomputing what your local forward would be into global coordinates.

A way to look at it is like this:

```
Vector3 arrow = transform.forward*2 + transform.right;
redCube.position = transform.position + arrow;
```

`arrow` was created by thinking about our local forward and right, but then it's just a regular vector3. When we add `arrow` in the second line, the system doesn't know or care how it got made.

You don't need to see the numbers, but it might make things clearer. Here's a simple program to show 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;
}
```

You should be able to select the object, press Play, hand-spin it in Scene view, and watch how those variables change.

With no rotation, these should look the same as global – `right` will be (1,0,0), `fwd` will be (0,0,1).

If you spin to face backwards `fwd` is flipped to (0,0,-1) and `right` will be flipped to (-1,0,0.)

Rotating to 45 degrees, so you're facing north-east, shows `fwd` as (0.71, 0, 0.71). X and z are equal, which seems right. The numbers look funny, but the length really is 1. You might recognize them as the sin and cosine of 45.

If you spin 45 degrees and then tilt back 45 on x, so your forward sticks up and northeast, it becomes (0.58, 0.58, 0.58). All three the same seems correct. And it really is length 1. They follow the hypotenuse rule: square them and add them up and you get 1.

Again, you don't need to know the numbers. But if you know there has to be trigonometry somewhere, and it was bothering you where, this is where. Or if you were wondering how our `transform.position` can store local and global together – now you know it only stores global. And if you see some of the numbers for local axes and they look odd – they're sins and cosines and are supposed to be odd.

2.4 Childing

We know that child objects lock onto their parents, tracking them as they move and spin. The way that's accomplished is by using the parent's local coordinates. If we play with children a little we might get a better feel for local axes and "local space."

You may have noticed that when you go to the Hierarchy panel, and drag one thing into another to make it be a child, the numbers for Position suddenly snap to new values. The object hasn't moved. The change is because it's now telling you the position in the parent's local coordinates.

The slot still says Position, but it's not. Here are some fun things to try with a child:

- Enter (0,0,0) for the position (which is really the to-parent offset.) The child will snap to the parent. That should make sense. 0 away from it in all directions is exactly where the parent is.
- Use the Inspector variable slide trick (mouse to in front of a box and the cursor changes to a double-arrow, letting you know you can slide the value.) If you slide z back-and-forth, the child moves along the parent's forward/backward.

  Try spinning the parent and trying it – changing z moves on the parent's new local z axis. Likewise with x and y. Sliding the child's x, y or z traces out each of the parent's local axis.
- Enter something not too hard like (2,0,0) for the child's position (it will move) and spin the parent. We already knew the child would spin with it, but now we can check the math. In any new spin we can find the parent's red x-local-arrow, and to will be pointing to the child, 2 spaces away.

- Drag a child from one parent a different parent. The numbers will snap again. Now it's in local coordinates from it's new parent.

  This is another way of seeing that, obviously, each object has it's own local space.

The system always remembers your real-world position. For children it also remembers an extra set – local from the parent. In code, we're allowed to use and set both, and the system updates the other. Suppose `cc` is a link to one of our children:

We can use `cc.position=` the same as always. It will move the object to that real-world spot. It recomputes the local position to the parent, based on where we put it.

The new fun part is `cc.localPosition` lets us directly set the from-parent numbers. It's a way to use raw local coordinates. If `redCube` is our child, this will put it directly in front of us:

```
public Transform redCube; // this is one of our children

void Start() {
  redCube.localPosition = new Vector3(0,0,2); // snaps to 2 in front of us
}
```

The back part fools you, since it's just raw (0,0,2). The key part is `localPosition`. It knows it's value is local coordinates from the parent. It's a little like how `transform.Translate(0,0,2)` knows to move in your forward..

Here's move-a-ball-under-me-from-back-to-front, rewritten to use a child and localPosition. It's a little simpler than before:

```
public Transform redCube; // assume this is a child of us

float zz=-3; // moves from -3 to 5

void Update() {
  redCube.localPosition = new Vector3(0, -1, zz);

  // same code as before, moving zz
  zz+=2*Time.deltaTime;
  if(zz>5) zz=-3;
}
```

Of course, using a child and `localPosition` to place objects is just a shortcut. For my child `redCube`, these two things are the same:

```
void Start() {
  redCube.localPosition = new Vector3(0,0,2);
```

```
  redCube.position = transform.position + transform.forward*2;
}
```

The thing is, they really *are* the same. In the first one, the system knows to add my position and to covert +2z parent-local into real coordinates. In the second, I do that math myself. The work is the same either way – making a child is just a time-saver.

The whole idea of "my local coordinates with me at (0,0,0)" is called Local Space. The easy way to explain what children's position numbers means is they're in the parent's local space. This isn't a new rule – just a shorter way to describe some things. If we're just using `transform.forward`, we'd call it your local axis. But if we're doing a full xyz position from us, it's nicer to say "in my local space." We'll see more tricks with it, much later.

2.4.1 `SetParent` notes

This section is a "while we're on the subject" thing. Unity has some interesting rules for setting your parent in code, and we may as well see them while we're on the subject. But there's nothing new here about local or vector math.

The basic command to make something your parent works the same as doing it by hand: nothing moves now, but you'll track your new parent. A simple use is if an arrow sticks into something. The arrow has hit, it's where it should be, and you glue it there in case the target starts moving:

```
arrow.parent = targetStuffedWithHay;
// or:
arrow.SetParent(targetStuffedWithHay);
```

To un-parent, set the parent to nothing: `arrow.parent=null;`. All normal objects are walking around with `null` for their parents.

Often you want to create a new child in a certain position. It's almost always easiest to combine steps: make it, child it, position it:

```
Transform newDogChild = Instantiate(dogPrefab);
newDogChild.parent = transform;
newDogChild.localPosition = new Vector3(0,2,0);
```

There's one rare oddball use of `SetParent` that you almost never need. It's a hack that lets you pre-set what you want your local position to be. You put you future local position in your position. Then the `setParent` command does all the work – adding `false` tells it to copy the current position into the new local position. It looks like this:

```
Transform newDogChild = Instantiate(dogPrefab);
// fake setting position. This is really our future local position:
newDogChild.position = new Vector3(0,0,2);

// "false" means to also copy our real position to be our new local position:
newDogChild.SetParent(transform, false);
```

The purpose for this is prefab UI elements – buttons and sliders. They're always children of a canvas or canvas panel, and they have a lot of tricky local settings in their RectTransform. This trick lets you preset local settings in a button prefab. When you Instantiate and child this way, you get the local settings you worked so hard to make.

But there's a better way. The instantiate command has an option to create something directly into the parent:

```
Instantiate(button1Prefab, canvasPanel2, false);
// false means to copy button1's position into it's localPosition
```

This is pretty nit-picky specific Unity stuff, but it shows a little more how you have to have a local position from something

Bonus terms: computer science and unity-slang use parent/child verbs backwards. In computer science (which is how many game programmers started out,) you use the word for what you are: `A.parent=B;` is *childing* A, or "childing A to B," or possibly "parenting B to A" (that's more awkward, but everyone would know what you mean.)

You'll often read Unity users writing how `A.parent=B;` *parents* A to B. I the computer science guys have been doing this longer and know the best way to say it. Be aware you might see it both ways.

2.5 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 did made a mistake.

If you incorrectly use `Vector3.up` instead of `transform.up` you might not notice for a while. You often only spin. Your left, right, forward and backward are all changing but your up is the same as world up.

Much later you might tip sideways and `Vector3.up*2 + transform.right` suddenly starts giving funny results, since it should have been `transform.up` the entire time.

If something expects local coordinate then using `transform.forward` gives wrong results. For examples, using it in localPosition or Translate:

```
child.localPosition = transform.forward; // ick. use Vector3.forward
transform.Translate(transform.forward); // ick. same.
```

The problem is this double-converts. In your mind you had "local (0,0,1)." These two commands like local and will convert it. `transform.forward` is only for when the comand doesn't like local and you need to hand-convert to global yourself.

Another example is with AddForce. this first one works the way you think. AddForce takes global xyz and `transform.forward` gives globals:

```
rb.AddForce(transform.forward*6); // fine, since this takes global axes
```

But this double convets:

```
rb.AddRelativeForce(transform.forward*6); // not good. double-converts
```

The point of using the `Relative` version is so we can write just (0,0,6).