# Chapter 0

# Intro

This is about the math for 3D positioning and rotating, and the computer code. There are examples and some tricks, but the goal is for you to know enough to figure out oddball move and spin problems for yourself.

The working examples run in the Unity3D game engine, written in C#. No special version, since no advanced features are used. You could probably read this if you don't use either – C# and Unity are fairly generic.

The examples assume you know the basics of coding, and basic Unity3D set-ups (you can place a script on a gameObject, know how to drag things into script Inspector slots). But they don't require much more.

## 0.1 Review of the basics

Before we start moving and rotating, it's nice to have a review of 3D basics and the choices Unity makes:

**xyz axes:** Different systems aim these in various directions. In Unity, y is up and down. x is the usual left/right, leaving z as forwards and backwards. A funny effect of this is a standard floor runs along x&z (not x&y as you might expect).

Positive/negative run in the obvious directions: +y is up, +x is to the right, and +z is forward (further away from you, if you're looking from the front).

For examples: 3D trees made for Unity have the y axis running from roots to crown. A 3D cow made for Unity should be facing along +z, since that's forward.

**Coordinates:** There aren't any special coordinate values. You can place things where-ever you like. For example:

- $y$ less than 0 isn't "underground" or underwater. The ground, water, lava, is wherever you put it.

- There's nothing special about negative values. For example (0,0,0) can be in the center. Half the map will be have negative coordinates, which is fine.
- There are no special out-of-bounds values. The world can go as far as you want, in any direction. Edges are where-ever you place them.

It's probably better to put things somewhat near (0,0,0,) just to keep the numbers small. But you can easily move things around later.

**Units:** The real world meaning of units is whatever you want. In other words, the 3 in (3,0,0) can stand for any real distance. For examples:

- If your game takes place on a board, 1 unit = 1 square is nice. Or maybe 10 units per square. That way you can make "big" pieces 9x9, leaving plenty of round numbers for smaller pieces.
- For a game taking place somewhere real, you often have real measurements in actual meters or yards. In your game, use that for your units. If you have a game about mice, and know real mice measurements in inches, use inches for the units.
- The 3D models you use aren't a good cue for scale. You often have a cow in meters, a duck in inches, and a shovel in "who knows?". Pick a scale first, then adjust the models to it.
- It's fine to not know the scale. Just place things where-ever they look good. If something forces you to compute the scale later, and it works out to some funny decimal, that's not really a problem.
- Officially Unity uses 1 unit = 1 meter. But that only matters for gravity. You may not be using gravity, and it's easy and common to change gravity's value. It's completely safe to not use meters for the units.

**Scale:** It makes sense to scale a cow 10% larger or smaller. But for a box or a log, being able to stretch it independently along x, y, or z makes more sense. It turns out that everything can stretch/shrink on x, and y, and z. To scale the whole cow 10% larger, you have to stretch it 10% on them all (just enter 1.1 in all 3 slots).

We always have the option of making our cow twice as wide (scale 2,1,1), even though it would look fake.

**Model origins:** Suppose we place a cow at one specific xyz. Which exact part of the cow will be there? The tip of the nose? The center? The feet? The answer: it could be any, depending on how the cow was made.

The "placement point" is officially called the model's *origin*. It's technically the (000) used when the model was created. We can't change it. It's usually in an obvious spot, but not always.

For example, trees and cows usually have their origins on the bottom. You can place them on the ground and have the whole tree going "up". Symmetric objects often have it in the center. A shovel might have it at the center of the handle, to make placing it in a hand easier.

## 0.2 Slightly less basic

**Rotations:** 3D rotations are shockingly complicated. We'll play with them more later, you won't need to memorize this:

The Inspector has rotation slots for x, y, and z. Imagine those are rods running through the origin of the model (yes, the origin also controls how an object rotates). Essentially, y spins us like a top, x rolls us forwards and z rolls us sideways.

Strange at first, x isn't a true-forward summersault. It's uses forward for the cow, after the y-spin. z is the same way. You can spin around the real x and z using the visual rotation tool, but you'll see x,y, and z all recalculated in the Inspector.

There's more: the Left-Hand Rule says which way counts as positive rotation: grab the axis for the rotation with your left hand, thumb in the positive direction. The way your fingers curl is a positive rotation. Try this for x and it tells you +x is a forward roll, meaning -x is a tilt backwards. For z this rule tells you +z rolls left, which seems funny, but the math won't work unless it does.

Basically, 3D rotations can seem simple at first. But once you start using them, you start needing more and more rules and seeing more weirdness. The trick, which we'll see later, is not to use x, y, and z.

**Common problems with real 3D models**

This section is for if you want to bring in some 3D models for the rest of the examples. That can be helpful, but there can be some fixable problems:

- They can be much too big. 3D modelling programs sometimes like to use 1 meter = 100 units. When you bring in a model that large, you may not be able to see it, since it completely surrounds you.
- They can be facing the wrong way. Some programs think y is forwards, with z as up. This can be confusing when we use commands that assume +z is the front.
- The origin can be somewhere we didn't want it. Maybe it's below the feet, but we'd prefer it centered.

**The parent trick**

This trick can fix everything wrong with a model:

First pick out an obvious space. For a cow, make a cow-sized +z-facing cow parking space for it to stand on. Then adjust the model to fit there. Spin it to face +z if needed, fix the scale so it's your perfect cow-size, rotate so it seems to be aimed on +z, and move it to fit the space the way you want, feet on the ground. Nothing special involved yet. This step is totally obvious.

Next create an empty gameObject. Put it where you want the origin to be: center of the cow, on the ground, at the cow's nose – wherever you want. Do not change this empty's scale or rotation. Only more it to your preferred cow-origin.

Finally, in the panel with the names, drag the real cow into the empty, making it a child. The cow model is now locked into that empty and will track it as it moves. We'll never touch the real cow again. We want to freeze those adjustments we made. The parent now counts as the cow

```
cow1 <- an empty gameObject, renamed cow1
  cow <-cow model. Adjusted the way we want, then moved inside of cow1
```

To see it working, move the empty around, spin it, scale it. It looks just like we're playing with our dream cow.