Chapter 0

Intro

Positions and rotations in the Unity3D game engine are about the same as they are anywhere else. One of the nice things about Unity is it steals the standard stuff everyone else uses, and doesn't need to make up many custom shortcuts. Learning Unity is close to learning the "real way" and vice-versa.

But this is about specific commands and how the numbers work in Unity, with sample programs as Unity `C#` scripts. The coding will be pretty basic, and limited to rotation stuff. For example, I'll show you how to make a simple y-spin, with an example where y increases. If you want y to bounce between 90 and 270 – that's just regular programming, so not mentioned here.

0.1 Review of basics

Before we start moving and rotating, it's nice to have a review of 3D basics and the rules Unity uses for them.

xyz axes: There are a few ways these can work. In Unity, y is up/down, which means the ground is x&z. X is left/right, Z is forwards/backwards. Or, think of x as east and west, and z as north and south. A "front view" in Unity (where the ground is on the bottom) is an x/y screen (sideways and up,) with +z going away from you along the ground.

Positive/negative goes the way you'd expect: +y is up, +x is right/east and +z is forward/north.

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, or water if you have it, is wherever you program it to be.
- There's nothing special or bad or complicated about negative values. If you use (0,0,0) as the center of your game world, half will be negative,

which is fine.

- There are no special out-of-bounds values. The edges of the board or the world are where-ever you program them to be.

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 units are whatever you want them to be, and don't have to stand for anything. For example:

- If you're using a game board, 1 unit = 1 square might be nice, or some other round number. Like making each square is 9x9 might make it easier to size small, medium and large pieces.
- If the game is real, like in a house or outside, 1 unit = 1 meter (or 1 yard or 1 foot) might be good.
- If you get 3D models from other places, the unit scales are usually all different – a 3 unit tall cow with a 350 unit tall barn. You'll have to resize them, no matter what units you use.
- It's possible to not even know the scale. You might position things in a play area, make the game, and it just so happens the area is 27.3 units wide. You don't need to know what they stand for to know half of that is the middle.
- Officially, 1 unit is 1 meter. But not really. That only matters if you use the preset value for realistic gravity, and no one does.

Model origins. If you know 3D models, Unity handles origins in the normal way. Parents solve problems the same as usual. If you don't, here's a summary:

You can use the arrows to drag a Unity Cube where-ever you need, and that works fine. But suppose you place a Cube just touching a wall at x=10. You'll notice the Cube has x=9.5. The premade Unity shapes are centered, and the Cube happens to be 1x1x1, which means it goes half a unit in every direction from where you place it.

In technical terms, the Cube's origin is in its center. When you position it visually, that won't matter. But when the code does it, which involves setting numbers, you need to know this stuff.

It so happens that "centered" isn't the rule. If you import a cow, for example, there's a good chance its origin will be down between the feet. That makes it so you can compute a spot on the ground, place the cow there, and the body will automatically go above the ground. If the cow's origin was centered, placing it on the ground would put half of it below-ground, like that Cube.

Each object has its own origin, which is decided when it was created. You can't change it later (but you can use the parent trick to fake-change it.) Imagine you're creating the cow. You'd find (0,0,0) in that program, then build the cow either around it, or up from there. Where-ever (0,0,0) is in that cow you made, which could be anywhere, that's the origin when someone uses it.

We don't really care about origins yet. We'll be using just a small Cube or Sphere that represents a point. If our code makes us go out to 10, our center should hit there, half will hang past, but we'll know it worked.

It's just something to be aware of. If you try the later stuff with a model you imported, the origin will track your numbers exactly, with the rest just being where-ever.

Rotations: 3D rotations can get pretty complicated. The basic idea is you can spin on your x or y or z axis.

Imagine you have a cow on the ground, facing +z (which is considered forward.) Y-rotation turns the cow (remember y is up/down in Unity, so it's like a twirling merry-go-round pole horse.) X runs left-right, so an x-rotation rolls the cow forwards. Z runs forwards/backwards, along the length of the cow, so a z-rotation tips the cow sideways, like it's on a barbecue spit.

Objects also rotate around their origins. Unity shapes will spin nicely in all directions, since they have centered origins. But take the bottom-origin cow. It will spin fine on y, but an x or z rotation spins the cow around it's feet. If it's standing on the ground, a forward roll will put it completely underground before coming back up.

This is still not really important for us. Just if you use an imported model and it appears to rotate funny, it's simply a non-centered origin.

For turning (y-rotations,) Unity thinks 0 degrees is facing forward, along +z, and positive degrees spin clockwise. 90 degrees is facing right/east.

In general, Unity thinks (0,0,0) rotation should be facing north, head up (like a cow standing on the ground.)

The rule for direction, for all 3 axis, is that Unity uses a left-handed coordinate system. You can look up some nice pictures of this. Here's a short version: wrap your left hand around the axis, thumb pointing positive. The curve of your fingers is the plus rotation direction. If you try it for y (like grabbing a lamp pole in front of you, thumb up) your fingers go clockwise.

The biggest surprise is z. Grabbing z with your thumb pointing forward is awkward - maybe pretend you're grabbing a railing next to you. Whether you grab it from the top or bottom, your left-hand fingers will be rotating counter-clockwise. If you spin on z to tip the cow, you'll rotate to the *left*. Tipping to the right requires a negative rotation.

If you know real trig, everything about Unity's system seems wrong. Unity does have built-in sin, cosin and so on, and they use 100% correct trig. But the

rest of the Unity rotation system is degrees, clockwise, 0=north. If you decide to mix trig and Unity-rotations, you'll have to convert back and forth.

Problems with real 3D models

Again, using a small Cube or Sphere for testing is fine, but bringing in a real model is fun, and you eventually need to do it for a real game. There can be problems when you do.

Most 3D modeling programs flip the axes. They use z=up and x&y as the ground. No special reason - they just do. If you import a cow, it may be backwards and facing up.

You can manually spin it the right way. But suppose a programs sets the cow to spin from 0 to 360. The first thing it does is snap to 0, destroying your fix and giving a weirdly spinning cow.

That's the real problem with an x-facing cow. It seems perfectly fine - it's facing where it's facing, and it can easily spin to face anywhere else. But Unity thinks +z is forward, and some program will eventually attempt to face the left-side of your cow (which is +z for a +x facing cow) to the target.

Standard sizes of modeling programs are also funny. They are naturally in the hundreds, just because. An imported cow will often be so large that you're entire game is inside it. Scaling it down to size 0.01 can be a pain, and falls off when a program tries to shift it between size 1 and 2.

The parent trick

Most real imported objects will need an adjustment to their size, rotation or origin. Even if something is perfect, you'll probably need another version of it with things shifted around. You can adjust these all at once using a parent. This is also how 3D modeling programs do it.

Suppose you have a wrong-size, wrong-facing, wrong-origin cow. To fix it, create an empty object, maybe named cow1. Leave it normal scale and no rotation, and place it in any spot you find helpful. Make the cow a child of it. Take that child cow and move, resize and spin it to how you want it. Then never change the child cow again.

Now the parent, cow1, acts like the cow you wanted. The advantage is it looks like you're not doing anything. cow1 can have no rotation and be size 1. The changes are safely hidden away in the child.

You can even do this again with the same cow. Parent cow2 can have the same cow with a different origin.