Chapter 0

Intro

This is about the math for 3D positioning and rotating, and the computer code. There are examples and some real tricks, but the goal is for you to know enough to figure out all of those oddball move and spin problems for yourself. This is about 70% regular math.

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

0.1 Review of basics

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

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. But a 3D cow made for Unity should be facing forward, meaning the z axis runs from it's tail through the head; with +y running from feet to back.

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 game takes place on a board, 1 unit = 1 square is fine. That way +1x is also +1 square to the right.
- For a game taking place somewhere real, 1 unit = 1 meter might be good. That way you can scale everything to actual measurements. If you prefer yards or inches (the game is about mice?), 1 unit = 1 of those is also fine.
- The 3D models you use aren't a good cue – they tend to have inconsistent scales. You might get a nice set of barnyard animals at 1 unit = 1 meter, then some buildings at 1 unit = 1 foot. They are easily rescaled, and almost always are.
- It's fine to not even know the scale, or even to have one. You can let the camera shows some area, and hand position things using the drag-tools. Everything is just where it is. If you find you need to know the numbers for something later, pasting them into code is fine. The computer won't care if cows are an oddball 0.741 units tall.
- Officially, 1 unit is 1 meter. But that only matters if you use the preset value for gravity (-0.98 on y). It's easy to change, and most people do.

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 if the floor is at 0 and your code puts a

centered cube there using y=0, it will be half-way embedded.

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 centered between the feet. Each object has its own origin, decided when it was created. The person making that cow found (0,0,0) in the blank screen. Then decided whether to build the cow all around it, or up from it, or maybe up and forward from (0,0,0) (which would make that cow's origin between its back hooves).

For now, it's enough to know that when we place a built-in Cube or Sphere somewhere, we're placing the center. But that if you grab other 3D models, they may intentionally not be centered.

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 on all axes, 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. One thing is knowing a funny-looking cow rotation simply means a non-centered origin (which you should be able to pick out if you watch to rotation for a few spins). Another is if we want a non-centered spin for a centered Cube, we merely need to change the origin (using a trick, below).

For y-rotations, Unity thinks 0 degrees is forward, along +z, and then it goes clockwise. So 90 degrees is facing right, 180 faces backwards (as you'd expect), and 270 faces left.

The rule for x&z spin direction is 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.

For x you'll be grabbing a sideways line, thumb aimed right (like you're showing someone a briefcase). Your fingers will curl up and over. This means a +x rotation tilts down into the ground. To tilt backwards, making the cow look upwards, use -x.

z is the funniest. Grab a line coming out from you, thumb forward (an awkward underhand, for me). You fingers curl up and left – meaning +z rotations

tips you to the left.

If you know real trig, this entire system of degrees and clockwise might seem funny. It's done to be simpler for Game designers, and all normal Unity API's use it. Unity's has built-in sin, cosin ... and it's all correct trigonometry. If you decide to mix trig and Unity-rotations, which you rarely do, you'll have to convert back and forth.

Problems with real 3D models

We'll only use Cubes and Spheres, which are fine for testing. But a real game will need some cows brought in, and they can have all sorts of problems.

3D modeling programs tend to use flipped axes from Unity3D: z=up and x&y as the ground. You can use a cow made that way, but Unity thinks the cow's back (which is the +z part) counts as forward. It will be on it's tail, feet facing us. We could fix it by hand-spinning. But the y-axis is still running the wrong way – if we try to turn the cow, it will barbecue-spin instead. For a moving cow, we need to fix the wrong axis.

Modeling programs also like to use 1 unit = 1 centimeter. An excellent, cheap cow-maker you hire will tend to give you a 220-unit long cow. You can hand-fix the scale; but you have the same problem if your code tries to tweak the size. Fun fact: most 3D models are see-through from the inside. That super-huge cow might go around your entire game, and you won't even be able to see it before scaling it down.

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. It drags around the real cow, keeping the adjustments you made. It appears to have the correct origin, and scale, and axes. The adjustments are hidden away in how the real child cow connects to cow1.

You can even do this to make other versions. A new empty, cow2, can have another copy of that cow inside, but with a different origin scale and axes (maybe for a menu-icon cow, which needs to be small and right-facing).