Chapter 21

Float problems

This is a short practical section about the way `float` numbers sometimes round funny, and how we usually handle it. All languages have these problems, and similar solutions.

It's not a basic idea of programming, but to write real programs you probably have to know it. I thought this was a good spot for a relaxing chapter.

21.1 The Problem

In regular math, we get lots of infinitely repeating decimals, for examples 1/3, most square roots, or trig like sin/cosin and PI. If we're solving equations for real, on paper, we carry them around, hope they cancel, and only compute them out at the end – like 1/3 times 3/5ths. But in computer math, we just round to the million's place, and that's good enough.

In a computer, one-third times 3 isn't 1. It's 0.999999. Taking `1.0f/3.0f` rounds to 0.3333333, times 3 is 0.999999. That's so close to one that no one could possibly tell the difference, but it's not exactly equal.

Computer numbers have one more source of rounding. They really store things in binary, which has a one-halves place, a one-quarters place, one-eighths and so on. In a computer, 0.1 (one tenth) is an infinitely repeating number, which is rounded to something very, very close to 0.1 (you can look this up for more.)

That sounds terrible, but it's really not. Any serious math already has at least one square root, or 1/7th, or something that's for sure a rounded repeating decimal. So we already have to assume all answers are rounded to the nearest millionth.

For example, this adds 0.1 to `n` every Update. It never hits exactly 1 – the `if` will never print:

```
public float n=0;
```

```
void Update() {
  n+=0.1f;
  if(n==1.0f) print("one"); // this never happens!!
}
```

We know it's *not* really adding 0.1 ten times. There's one exact thing that 0.1 rounds to in binary, and it's adding that 10 times. It hits a number very, very close to 1.

Some more funny rounding: `10*0.1f` is equal to 1 (the compiler changes it to 1.) But `n*10`, when `n` is 0.1, does not equal 1.

Any little change in the math can cause rounding to change. So it seems random – almost like the computer puts +/-0.000001 on decimal math, just for fun. That's not true, but it's a safe way to think of it.

The computer doesn't have any problems with whole numbers. `float n=1.0f; n+=2.0f;` will always give you exactly 3.0. For example, you may be changing your position using only whole numbers. Checking `if(transform.position.x==7.0f)` is safe.

21.2 Solutions

These are all things that might make sense, or might only make sense when you have a problem where they apply. None of them are rules – just tricks to handle off-by-0.000001 problems. I'm not going to be using any of the later on.

Compare using "close enough."

Say we start at 0 and keep adding 0.1. We know the number will always be very, very close to 0.1, 0.2 . . . 1, 1.1 and so on, but will probably never hit those exactly. Suppose we want to do something when it hits 3.0 and something else at 8.5.

We can check for when it hits them within a rounding error:

```
// are we within a rounding error of 3:
if(n>2.999f && n<3.001f) {

// within a rounding error of 8.5:
if(n>8.499 && n<8.5001) {
```

Any rounding error is tiny, so this will only catch things that were supposed to be 3. For example, counting by 0.1 the number before 3 is 2.9. There's no possible way rounding errors could increase it to 2.999.

This is also what the `closeEnough` function from before is meant to do. For real we'd check `if(closeEnough(n,3))` or Unity has a built-in `Mathf.Approximately` which is meant to do the same thing.

Don't worry about it

Suppose your program moves `n` up from 0, at various changing speeds, and you want to know when it crosses the edge at 6. Checking `if(n>=6)` is fine. In theory the real math might hit 6.0, while rounding causes it to be only 5.9999 – it will take an extra step to go past. But you don't care; you weren't counting steps anyway.

Or suppose the speed is always 0.1. Now it should clearly take exactly 60 steps to move from 0 to 6. Rounding down might cause it to incorrectly take 61 steps. But, again, you weren't counting and don't care. It might look better with 61.

Something to note is the rounding is not random. The same math will always give the same numbers. If it falls a little short due to rounding and takes an extra step, it will do that every time.

Account for rounding in your math

If we're really worried about a possible extra step, adjust the checked number to account for rounding. This is a safe "are we past 6":

```
if(pos.x>5.999f) // are we past 6 with a rounding
```

It's the same logic as the "within a rounding error" test. This will catch any rounding errors on numbers which should have been 6 or more, but will never catch a badly-rounded 5.9.

Going further, suppose we step back and think about when we should count as hitting the edge. What if the last step normally overshoots by most of the move, like 5.98 to 6.08? It might make more sense to count as hitting the edge when we're within half a step:

```
if(n>6-mv/2) { // do hit the edge stuff
```

If we like that plan, the only trouble with rounding happens when we don't care. If half of the the next move would be take us to the border, it's all the same whether to stop now or take an extra step.

In general, the trick is to watch for cut-offs and think a little about them. Often what you want isn't proper math, anyway. Suppose you're computing the length of a health bar. It looks nice to have a last little sliver hang around, even if the real math would round to 0 pixels when you have health left.

Use integers

If you're storing money, use pennies – store $6.27 as 627. Divide by `100.0f` when you print it.

If you're storing gallons and counting by quarters (1.5 gallons, 3.75 gallons) then store it in whole number quarts.

If you have a score that goes up by 0.2, multiply everything by 5 and use ints. Divide by `5.0f` when you print it.

If you have something funny like sevenths, you can use two int variables – whole numbers, and sevenths. You'll have to use a few `if`s to handle sevenths not between 0 and 6:

```
sevenths += moreSevenths;
if(sevenths>=7) { sevenths-=7; num+=1; }
```

Force it to use integers

There are times when it's not naturally an integer, but you can sort of turn it into one.

For the mover, from 0 to 6 by 0.1, it should take 60 steps. I could rewrite it as an int counting from 0 to 60, like this:

```
public int steps=0;

void Update() {
  steps++;
  if(steps>60) steps=0;
  if(steps==30) {}// checking when we hit 3.0 exactly
  pos.x = steps/10.0f; // <- convert steps to actual position
   ...
}
```

The advantage is that my checks can use `int`s, so are perfect. The minor drawback is I have to remember that step 29 is really position 2.9 (or worse, if the range doesn't start at 0.)

Occasional resets

This goes with of the "don't worry about it" trick. If you're using rounded floats, adding a little every Update, you can get more and more off over thousands of additions. That might not be a problem. If it is, it can help to snap to a known value every so often.

An example is the back-and-forth platform. Suppose we add 0.1 from 0 to 6, then flip the speed and subtract, over and over. The rounding will probably cancel out, but it might not. The first time, we may hit 5.999, then the next trip we hit 5.998. Over a hundred trips (which might be a few minutes of time) we can get pretty far off.

If might just look funny on the screen, or we might jump over some important `if`. That's why I like to reset when I hit an edge:

```
pos.x += xSpd;
if(pos.x>5.999f) {
  xSpd=xSpd*-1;
  pos.x=6; // <- resets all rounding
}
```

The rounding errors only have time to build up over 60 moves, which is still a tiny amount of error. `pos.x=6;` starts us fresh.

21.3 < vs <= in float compares

Because we know there's rounding in floats, most people are pretty sloppy using `<` or `<=` when comparing them. For example, take `if(x<0)` when we know `x` is slowly moving down. Because of rounding, 99.9% of the time it will be a little above 0 in one step, then a little below 0 on the next. The odds of it hitting zero exactly are crazy small and also semi-random, so `<` or `<=` will almost never matter.

And if we were trying, we'd have `if(x<0.001f)` anyway. That 0.001 is just a safe number to account for rounding. We just as easily could have picked 0.0005 or 0.002. It's often just simpler to use `<`, since it's shorter and we don't think it will matter anyway.

But this doesn't mean we never care. If you subtract 1.0 from 10.0 over and over, you'll hit 9.0, 8.0 ... exactly. So `if(x<=0)` will for sure happen when it hits exactly zero.