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: 1/3, most square roots, or trig like sin, cosin and PI. We carry them around – our final formula will have square roots and division by 3. As we work it out, some will even cancel out. When we need a final decimal number, we work it out once, from the final equation.

Computers can't do that. Everything needs to be immediately converted into its decimal form. `float a=1.0f/3.0f;` has to immediately round down to 0.333333. This results in more rounding errors and some weirdness. For example `a*3` isn't back to 1. It's 0.999999. If we take the square root of 12, then square it later, we have the same problem – we almost get 12.

Computer numbers have one more source of rounding. They store everything in binary. Binary numbers have a one-halves place, a one-quarters place, one-eighths and so on. In binary, 0.1 is an infinitely repeating number. Many decimals that are fine for us, are repeating and rounded in the computer.

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!!
}
```

It's adding a number which is very close to 0.1. After ten times, it's very close to 1, but isn't 1. It's not really an accuracy problem. The main problem is that our `==` don't work. Other weirdness: `10*0.1f` is exactly 1. But `n=0.1f; n*=10;` isn't.

The end result is that computer rounding can seem random. Almost the same formula could give the correct result, or could be off by 0.0000001 higher and lower, depending.

To be clear, this is only a problem with rounding decimals. The computer doesn't have any problems with whole numbers. `float n=1.0f; n+=2.0f;` will always give you exactly 3.0.

21.2 Solutions

These are various tricks to handle off-by-0.000001 problems. These are when you have `if(n==10.0f)` and logically `n` should hit 10, but it hits 9.999999 instead.

Compare using "close enough."

We can fix the Update loop above by checking whether `n` is very close to 1:

```
public float n=0;

void Update() {
  n+=0.1f;
  if(n>0.9999f && n<1.0001) print("one");

  if(n>4.4999f && n<4.5001f) print("4.5");
}
```

The rounding errors are tiny so checking within 0.0001 is safe.

Doing the math inside of the `if` is a pain. This is why closeEnough functions exist. Unity has one built in (we also wrote one, way back):

```
bool closeEnough(float a, float b) {
  float diff=a-b; if(diff<0) dif*=-1;
  return diff<0.0001f;
}

void Update() {
  n+=0.1f;
  if(closeEnough(n, 1.0f) print("one");  // ours
  if(Mathf.Approximately(n, 4.5f)) print("4.5"); // built-in
}
```

This is a brute force approach. You could go through a program and replace every float `==` with a close-enough and it would probably fix everything without breaking anything new.

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 steps.

Something to note is the rounding isn't 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

Sometimes we can 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, compensate for rounding
```

Even more than that, we might decide we should stop when we're within a 1/2-move of 6. Our adjusted code is `if(pos.x>6-mv/2)`. The exact rounding won't matter any more.

Use integers

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

Say you're storing and printing gallons with fractions. But you only make them using quarts and cups. Since those all go evenly into each other, you can store everything in whole cups. Or you may be able to store everything in teaspoons, or half-teaspoons (as in `int halfTeaspoons;`.

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 `ints`, 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 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.