Chapter 36

Efficiency

In many of my examples, I make a big deal about how to make the program look nice. I don't say much about how to make it more efficient – in other words, making it run faster.

As you write code, there are a lot of little "this way or that way" decisions. For a bunch of different reasons, if you make all these little decisions based on what seems like it would run faster, it doesn't work and turns the code into a mess.

There are things to do to avoid super-slow code, but most of them are when you make the plan – not when you're writing each line.

Some notes:

36.1 Most speed-ups don't work

A lot of the things that seem, for sure, like they'd speed up a program just don't work. They do nothing, or make it run *slower*. Here's a list of why some things don't work:

The compiler rearranges things anyway

Writing an equation on one line or on several lines doesn't matter. The computer breaks it into step-by-step anyway. Suppose you have this:

```
x = a*b + c*d;
```

The compiler changes it into:

```
int temp1 = a*b;
int temp2 = c*d;
x = temp1 + temp2;
```

By the same rule, declaring lots of extra temporary local variables doesn't hurt you. The computer was going to fake-declare them anyway.

`if` expressions are the same way. For real, the computer can only have one thing in an `if`:

```
// you type this:
if(n>=1 && n<=10)

// compiler turns it into:
if(n>=1)
  if(n<=10)
```

Or's are broken up using a different trick (it's hard to show, but it works.)

The compiler might even flip the test. If you write `if(n>0) A else B` the compiler might turn it into `if(n<=0) B else A`.

So go ahead and write `if`'s and conditions how-ever it looks nicest. If there was a faster way you could have re-arranged them, you may as well let the compiler find it.

The compiler knows the speed-ups

Over the years, we've found a lot of mini-tricks to rearrange code to speed things up. And we've programmed them, and put them into compilers. In other words, if you can look up a speed-up trick, and it works, it's probably already in the compiler.

The computer will pre-do math with constants. For example `n=7*24;` is fine to get hours in a week. The compiler turns it into 168. When the program runs, the line is just `n=168;`.

In general, leave math in whatever form is easiest to read. I like to write `Random.Range(1,6+1);` to remind me it's going to 6, with the 1-less rule. It's not slower, since the compiler makes it a 7 before the program ever runs.

The compiler can re-arrange to factor out common terms. For example:

```
int n = x+b*3-1;
int m = (y+b*3-1)/2;
```

You might notice both use `b*3-1`. We could compute that once and re-use it:

```
// "improved" version:
int q = b*3-1;
int n=x+q;
int m = (y+q)/2;
```

But a good compiler does this for us. Over 50 years very smart people have written papers about simplifying equations, which made their way into compilers.

If the code would look better, go ahead and precompute. Otherwise let the compiler handle it.

Speed-up logic is hard

This is an old example of a not-good attempt to speed up. Maybe you can see it quickly, but it gives the idea:

Part of resetting a game round might run `health=100;`. But it's possible we ended with 100 health – maybe we ran out of time, or fell off the edge. So we improve things with `if(health!=100) health=100;`

If feels like we're saving time by skipping lines we don't need to run.

But `if`'s take time to run, too. This new way is 1 or 2 steps, usually 2, compared to always taking 1 step. Our speed-up is really a slowdown.

But maybe we also have to reset the heath bar. Now a 1-step `if` could save us 10 steps. But we probably only die with 100 health one time in a hundred, so this still runs slower on average.

36.2 Premature Optimization

Let's suppose we can write each part of the program in two ways: runs faster, or easier to read – one or the other. Another way to say the same thing, suppose we can rewrite any part to be a little faster, but more confusing.

When would we decide to do it which way?:

- If we need to test it and are pretty sure there will be some things that need fixing, easy to read is better.
- If we're going to expand it later (for example, a match game using colors and we probably want to add matching shapes) then easy-to-read is better.
- Lots of code is temporary or experimental. Our first follow camera is good enough for now, but will probably be heavily modified after testing. Drag-over code might highlight borders of what it goes over, for now, and we'll think of something better later. Easy to read is better for any code we're not completely finished with.
- If we get a lot of requests from customers for changes in this part, easy to read is better.

- If this part is done, tested and you're confident it won't need to be changed, running faster is an option. But since we write everything as easy-to-read at first, we'd have to take the time to convert it, and we may have better things to do.

An example, here's my simple "keys move left and right" code:

```
if(Input.getKey(KeyCode.A) pos.x-=0.1f;
if(Input.getKey(KeyCode.D) pos.x+=0.1f;
pos.x=Mathf.Clamp(pos.x, -7, 7); // runs even if we don't move
transform.position=pos; // ditto
```

When we don't move, this code uselessly makes sure `pos.x` is between -7 and 7, and uselessly resets our position. We could speed it up by only running the last two lines after we moved:

```
float oldX=pos.x;
if(Input.getKey(KeyCode.A) pos.x-=0.1f;
if(Input.getKey(KeyCode.D) pos.x+=0.1f;
if(pos.x!=oldX) {
  pos.x=Mathf.Clamp(pos.x, -7, 7);
  transform.position=pos;
}
```

This runs slower! When we're moving, this is two lines *more* than the old code. It runs a little faster when we're not moving – but who wants a game that speeds up when you stop?

But I can keep trying. Here's my next attempt which really is faster:

```
if(Input.getKey(KeyCode.A) {
  pos.x-=0.1f; if(pos.x<-7) pos.x=-7;
  transform.position=pos;
}
else if(Input.getKey(KeyCode.D) {
  pos.x+=0.1f; if(pos.x>7) pos.x=7;
  transform.position=pos;
}
```

This always runs faster, but I think it's harder to read (two lines setting position, extra 7's.)

But wait. We weren't done making the game. We want to slide sideways for a few frames after you let go of a key. Then we want wind and bumping to also move the player. Those are going to be total rewrites. Time spent speeding up the version above was just wasted.

36.3 Not all code sections are equal

Suppose you have one player and thirty enemies (or anything using a script with Update.) For speed, the enemy code is 30 times more important. In other words, if you think of 30 little ways to speed up the player's code, you could have thought of just one way to speed up the enemy code.

Suppose you have a hundred lines in a script, then a nested loop around two lines. Maybe it checks every square in a 100 by 100 grid:

```
do stuff #1
do stuff line #2
...
...
...
do fortieth thing
...
do fiftieth thing
...
do one hundredth thing

for(int i=0;i<100;i++)
  for(int j=0;j<100;j++) {
      loop line #1
      loop line #2
   }
```

The inside of the loop runs 10,000 times. Compared to the hundred lines before it, the loop is 99% of the total time.

Making those hundred lines ten times as fast would be less than a 1% speed-up. Improving the lines in the loop is all that matters.

You can also have a huge loop even if you don't have that many items. Suppose we have an array of 100 foods and compare every one to every other. That's another ten thousand times loop.

Suppose the inside has a function call that also loops over the array. Everything in that function loop runs `10,000*100 =` a million times. From just a size 100 array.

Accidentally writing a triple-nested loop isn't all that uncommon. A lot of making code run faster isn't trying to speed up good code – it's looking for places like unnecessary triple loops.

But even more, there are lots of places where you don't care about speed at all: the Start / Pause / GameOver screens. Scrolling text. The code to fix player names longer than 20 characters. Calculating your "rating" after you finish a level (there's a delay and a little spinning animation, anyway.)

36.4 Is it already fast enough?

There are places where obviously we care about speed, but once we get it fast enough, there's no point getting any faster.

Almost any menu-based web program just has to run faster than human reaction speed, which isn't very fast at all. Humans can't click buttons more than ten times a second. If you can respond in 1/10th of a second (which is forever to a computer,) any faster won't matter.

Screens update at 30 or 60 frames/second. No matter how fast all of our Updates run, the picture still changes every 1/60th of a second. If we finish faster than that, the computer just sits around waiting.

For something with hundreds of enemies, all running scripts, squeezing it into 1/60th of a second might be a challenge. But a matching game will never be close to the limit. It's like mail is picked-up at noon, and we're rushing to put it in the box before dawn.

A typical menu has a Submit button where it makes a big database request, or waits for something over the web. There will be a 1 or 2 second internet pause. Most people will take a sip of a drink after they press it. Our code being a few milliseconds faster or slower won't be noticable.

36.5 Will it cause more errors?

Harder-to-read code probably makes it harder to spot errors, but beyond that, some of the known tricks to speed things up cause other complications that can lead to bugs. This is a fun example using the caching trick.

If you remember, the trick is to find some long, slow thing that you recompute every Update. Instead compute it once at the start in a global. This precomputes some other Cube's `Material` so we can easily change it's color:

```
public GameObject otherCube;
Material otherCMat; // shortcut to Material of otherCube

void Start() {
  // pre-compute ("cache"):
  otherCR = otherCube.GetComponent<Renderer>().material;
}

void Update() {
  ...
  // otherCube.GetComponent<Renderer>().material.color=c2; // long way
  otherCMat.color=c2;
}
```

The new way is a teeny bit faster (depending if we change the color a lot.)

But suppose the program sometimes changes where `otherCube` points. Now we have to remember to also change `otherCMat`. This speed-up gives the potential for a weird-looking, hard-to-find bug.

36.6 Things that work

In general, focus on high-use parts of the code - for example things that run for every game tile, every frame.

A main way of speeding things up is getting rid of accidental nested loops. They can be hidden, like a loop calling a function which runs another loop. A common Unity example is `GameObject.Find`. That's a loop through everything in the game – maybe a few hundred things and not too bad. But if you forget, you might put in inside another loop and it blows up into tens of thousands of steps.

`List`'s are a good way to have accidental nested loops. I mentioned they're really arrays and doing anything not at the back involves a slide-loop. If you don't know this, you might add to the front, `L.Insert(0,w);`, in a loop. Adding 100 items that way is 5,000 steps, compared to `L.Add(w)` at only 100 steps. Remove is the same way. Removing the last item is 1 step, removing the first is a slide-loop to shift everything down.

Other tricks involve looking at your plan.

For example, code handling monsters often checks line-of-sight to the player, and `Raycast` is doing lots of math. It's fine to run that maybe twice/second. Monsters will appear to have a slight delay before they see you, which is better, and you get at least a 10x speed-up over checking every frame.

Path-finding (computing a path through a maze or building) is crazy slow. You can often run that every few seconds, or only when something changes; and have some monsters just follow others.

One trick is keeping lists sorted. That can turn some double-loops into single ones. But it takes lots of practice and some theory.

LinkedLists are the other way of storing items in a row. They're very fast to insert and remove from any position, just as fast to loop through end-to-end as arrays, but much slower than arrays if you need to jump around with look-ups. Choosing an array vs. a linked-list can be a big speed-up, but knowing them also takes lots of practice.

Using a lot less space will speed up a program. Memory is divided into high speed, medium speed and regular. The more of your program which fits into the high and medium speed, the faster it runs.

This is partly why Unity uses floats instead of doubles. But it's still a matter of finding the worst ones. If you have one especially huge array, cutting the size in half might help; but don't waste much time on all the smaller ones which only add up to 5% of that big one.

The last things are making little tweaks to high-use code parts. This is stuff you only do if the game is done, you're out of good ideas, and you need to be maybe 2% faster.

For example, `v1.magnitude` (length of a vector) uses a square root. Unity provides a slightly faster version, `v1.sqrMagnitude`, which computes the square of the distance. It's harder to use and more error prone. But changing a few in the highest-use part of the code might give a 0.1% speed-up.