Practical Profiling

Handout for practicum 1 for INFOMOV
- updated for the 2019/2020 edition.

Universiteit Utrecht

Introduction

A crucial step in the optimization process is profiling. This document takes you through the basics of measuring application performance.

Situation

Our starting point is a working application, and for the purpose of this document, we will assume all intended functionality has been implemented. A minimal performance indicator is shown in the top-left corner. It fluctuates between values close to multiples of 16, which is caused by the use of *GetTickCount* for performance measurements: this timer has a 16ms resolution.

Goal

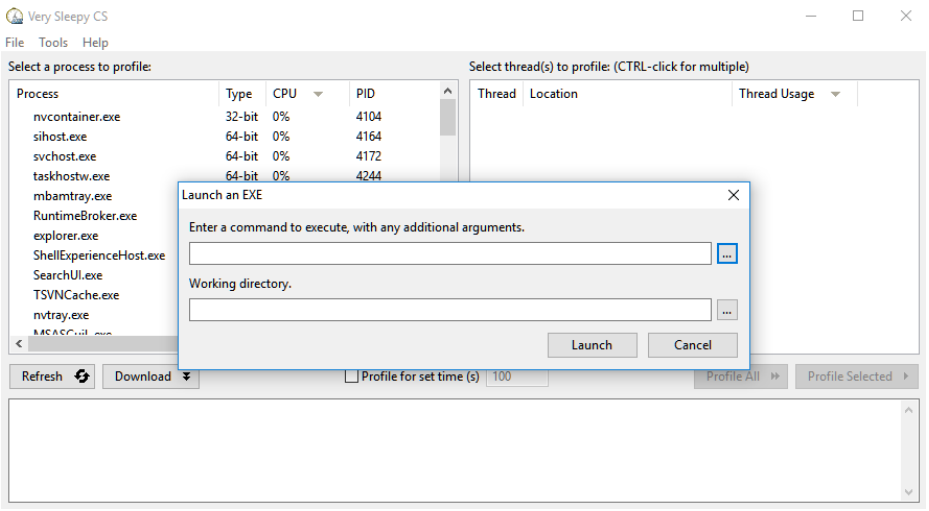
Our aim is to gain insight in the performance of this application:

- which sections take most time to execute;
- how does performance change over time.

This information will allow us to focus optimization effort on just the expensive parts of the code.

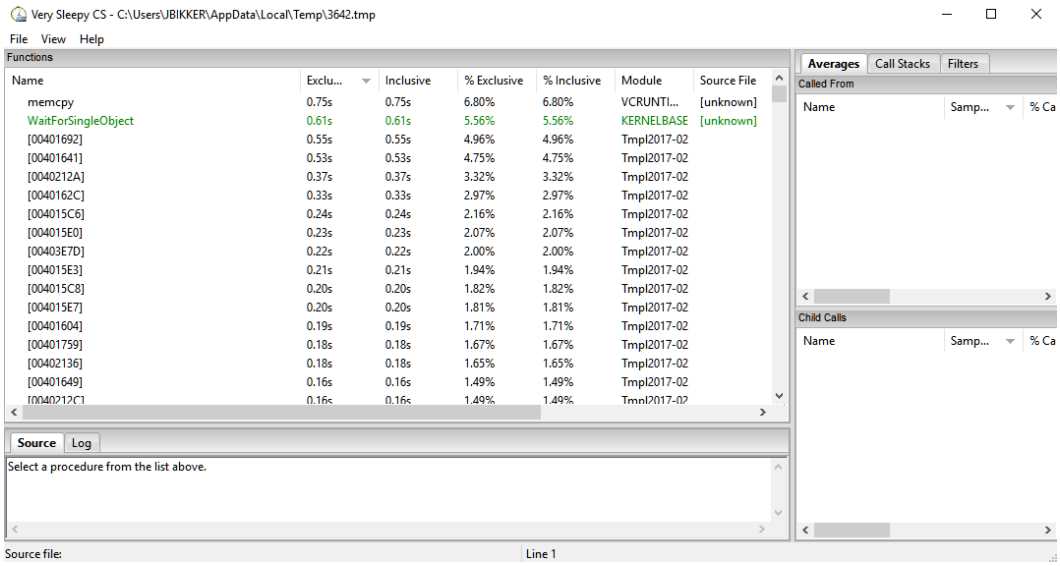
Very Sleepy

We can directly use the profiling tools supplied with Visual Studio. However, their functionality differs greatly between versions, and some important functionality is missing from the free versions of Visual Studio. Luckily, a free alternative exists: *VerySleepy*. Run the tool, and select **Launch** from the file menu:

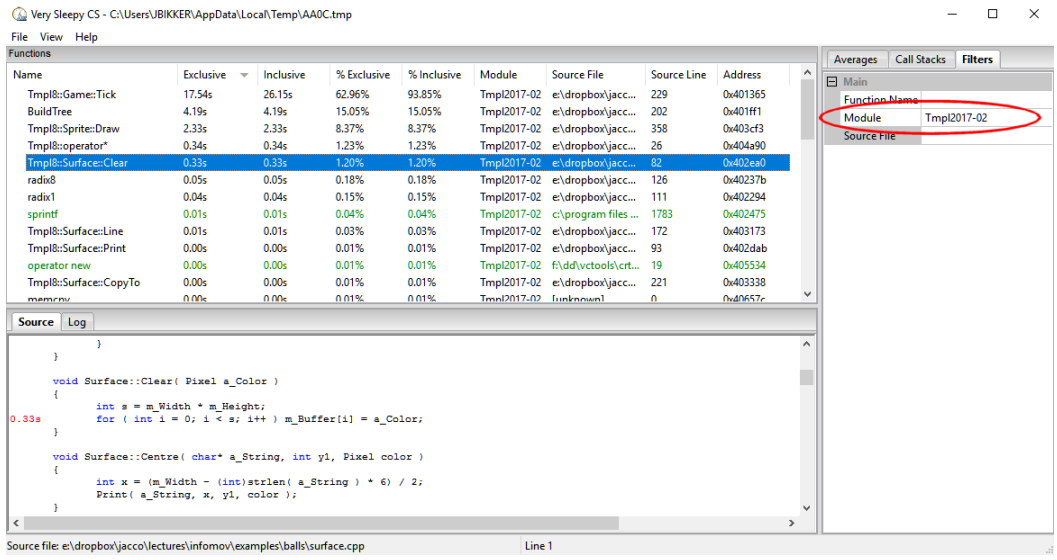


The executable to run is in the project folder of the balls application. This is also the working folder; make sure to specify this as well. Then click **Launch**. Make sure you profile the *release* version of the code: the debug version will be much slower, and may have different performance characteristics.

An initial run provides us with a rather useless overview:



To improve this, we need to include *debug information* in the executable. Consult the Lecture 1 slides for details on how to do this. After this change, we get the following output:



Note the filter on the right: we are not interested in any code outside our executable. Also note that VerySleepy is not just able to extract function names from the executable, but also the *full source code*. Let this be a warning: never distribute an executable with this information!

A few words on the contents of the columns: an exclusive column (seconds, percentage) shows the time spent in the function. Here, 62.96% of the time was spent in Tmpl8::Game::Tick. The inclusive column includes time spent in functions called by Tick. Only the values in the exclusive columns thus sum to 100%.

The source window provides even more detailed information. Some lines have an indication of the time spent at that line. Not all lines have this information: VerySleepy gathers its information by

periodically checking at which address the CPU is executing code. The time for a line is thus always an estimate, based on the number of times VerySleepy polled that line. Running the application longer will increase the number of lines that have an estimate, but in general, lines without a time estimate may be considered to be not very expensive.

Interpretation

Based on our measurements, we can conclude that the Tick function is by far the most expensive function (even exclusive time spent in called functions). Especially the section under "verlet: satisfy constraints" takes a lot of time. Why a line like `const vec3 d = p[idx] - p[i];` takes so much time is a topic for later, but one thing is clear: we should not be focusing on Sort(), nor on BuildTree, at least for the moment. A 35-line section in Tick is all that matters.

Performance over Time

While VerySleepy can report overall application performance and pinpoint bottlenecks, it cannot tell us how application performance is over time. It is for example perfectly possible that the Sort or BuildTree functions dominate frame time for one or two frames, even if they take little time on average. Or perhaps 'satisfying constraints' is far more expensive in some frames.

To get more insight we can instrument the code. The template provides a convenient high-resolution timer, simply called 'timer'. Two methods are of importance to us: reset() and elapsed().

We add a timer to the start of Game::Tick:

```
timer tm;
```

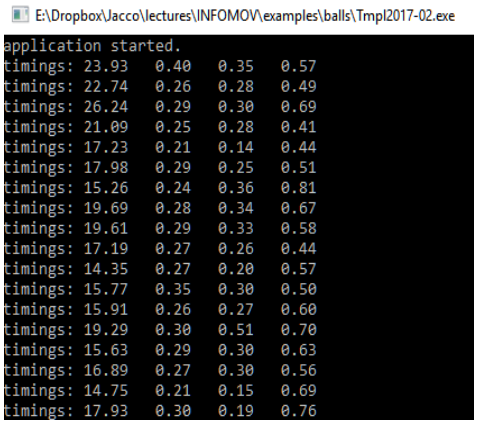
Now, we can obtain a split time in milliseconds at any time:

```
float measurement1 = tm.elapsed();
```

After gathering a few measurements we can report the timings. It is important *not* to report while timing: reporting may have considerable overhead. So, at the end of Game::Tick, we add:

```
printf( "measurement 1: %f\n", measurement1 );
printf( "measurement 2: %f\n", measurement2 );
```

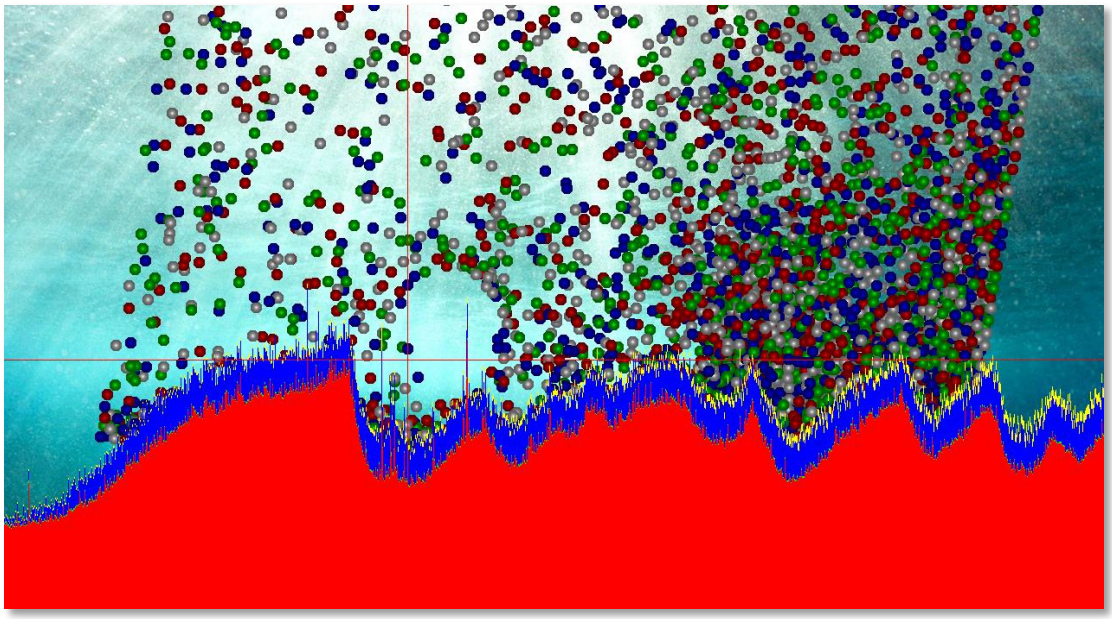
Some results are shown below to the right. The four columns are for satisfy constraints, sorting, render and finalize.



A few things pop out:

- The first frame is expensive.
- Satisfying constraints quickly goes down: after a second or two it is down to 10ms. Then, after a few more seconds, it goes up again.
- Sorting quickly settles for 0.21ms.
- Rendering starts low, but takes almost 4ms once all the balls have entered the screen.

Watching the text roll by is somewhat inconvenient. We can improve this by drawing a simple graph.



Here, red is the time for ‘satisfy constraints’ (10 pixels per millisecond), green the time for sorting, blue the time for render and yellow the time for finalize. The two dips happened when the application was interacted with, causing a ‘splash’.

The graph makes performance over time very easy to interpret:

- Start-up is slow, but it gets worse after the first 200 frames;
- Rendering starts fast but converges to a stable time slice;
- Some frames have significant peaks.

Based on this information, it is probably worthwhile to investigate *why* satisfying constraints is cheaper for some frames: perhaps it is possible to find a better algorithm that meets or beats the best time in all situations. This means we have high-level optimization work to do, before we get to low-level optimizations.

TODO

To gain additional insight, we can add additional graphs for the usual suspects:

- count the number of expensive operations (division, square root) to see if there is a relation with raw performance;
- count the number of memory operations (array accesses) to see if the application is held back by those;
- count the number of traversal steps in the binary tree;
- count the number of collisions.

In each case this helps us to identify what consumes time: if e.g. the number of tree traversal steps has no relation to performance, it makes no sense to focus optimization on this part of the code.

Get Active

Take some time to practice the material presented in this document, in case you haven't done so already:

- Replicate the VerySleepy results to make sure the tools are working for you and to get familiar with the workflow.
- Add the high resolution timer. Figure out what the resolution of this timer is, i.e. what is the smallest amount of time for which you get accurate results?
- Implement the graph, and operate the application to see the impact on various parts of the code.
- Add additional graphs to increase your understanding of what affects performance in this particular application.

In case you are not familiar with the template that is being used here:

screen->Line(x1, y1, x2, y2, color) can be used to draw a line for the graph. The screen is 1024x640 pixels, so the last line is at y=639, or simply SCRHEIGHT - 1. Similarly, screen width is stored in SCRWIDTH.

Colors are specified as 32-bit ARGB. Useful colors are (hexadecimal) 0xFF0000 for red, 0x00FF00 for green, 0x0000FF for blue and 0xFFFF00 for yellow.

The End

Questions and comments:

bikker.j@gmail.com or room 4.24.

INFOMOV 2019