COIS 2300H Lab ~~9~~ (10 for WI2020): Profiling, Tracing

**Preamble:** This lab is an effort to bring together concepts in the course. Thus far, as software developers you’ve largely limited yourselves to programs that do one or two conceptually straightforward things, and overwhelmingly the execution time of your program is easy to understand. But real software is rarely like that, even \*this\* software, that you’re looking at right now to view this document has hundreds or thousands of possible functions that could be executed at any given time and each of those will have performance characteristics that depend a huge number of factors.

Profilers come in a lot of varieties, and because they work different ways they have different characteristics. Which ones are right for whatever tasks you work on I will leave to you to discover as you plod through your careers.

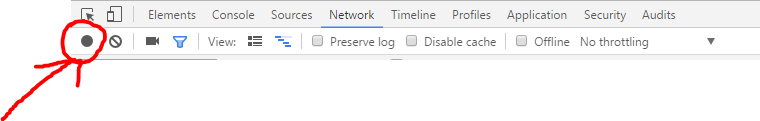
But we can do a couple of simple interesting things.

Mostly we’re going to use visual studio, but before we do that, lets do something really easy.

Open up Chrome (or Firefox). Open up the developer console (ctrl + shift + i on both)

Click network

Click the little recording icon



Now go to Amazon.ca

Click the recording icon again (it should be red this time).

Hey look at that, if you sort by (for example) time you can see how long all of the different page elements take to load!

Don’t dwell too much on it right now, but the idea is fairly simple – you can get a profile which functions and assets take how long and why. And then optimize your code/website/etc. around the largest problems.

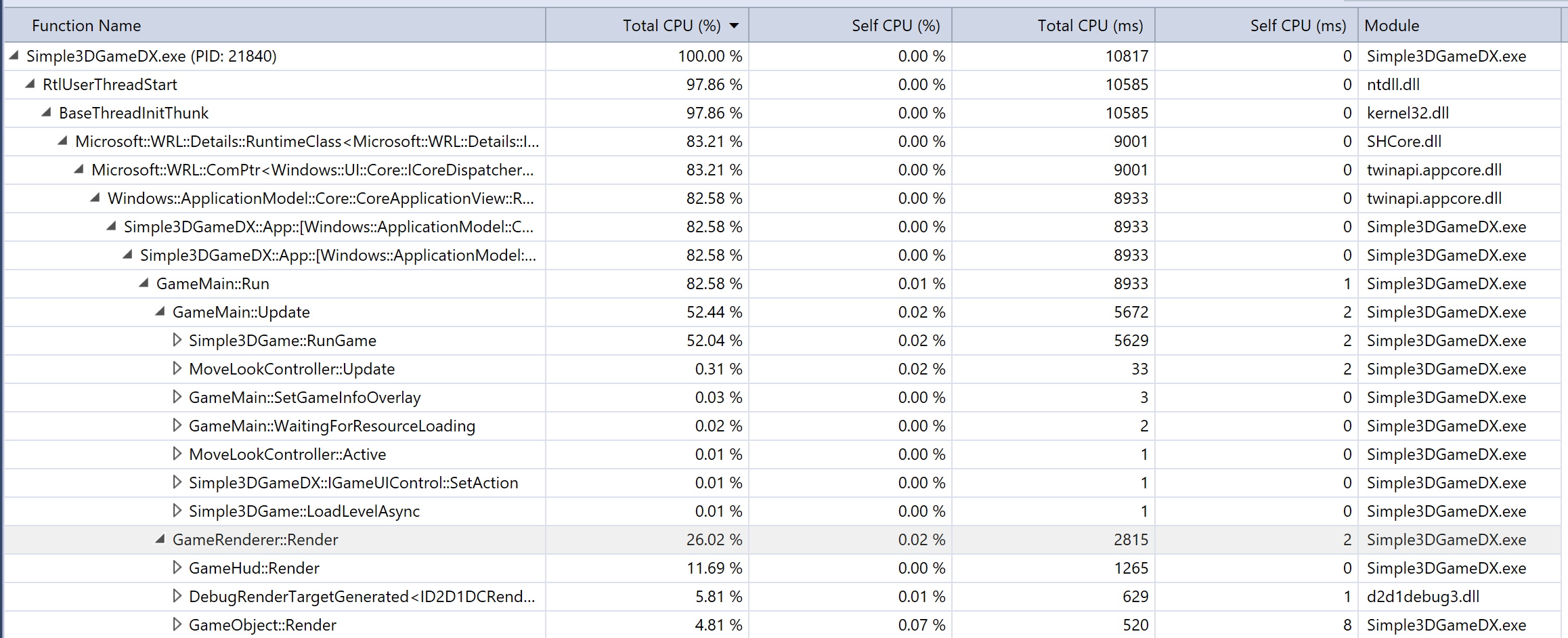
But ok, to Visual Studio where we write code

You can do this with **any program** you’ve already written (e.g. a 2020H assignment or one of the simple C++ programs you wrote for me).

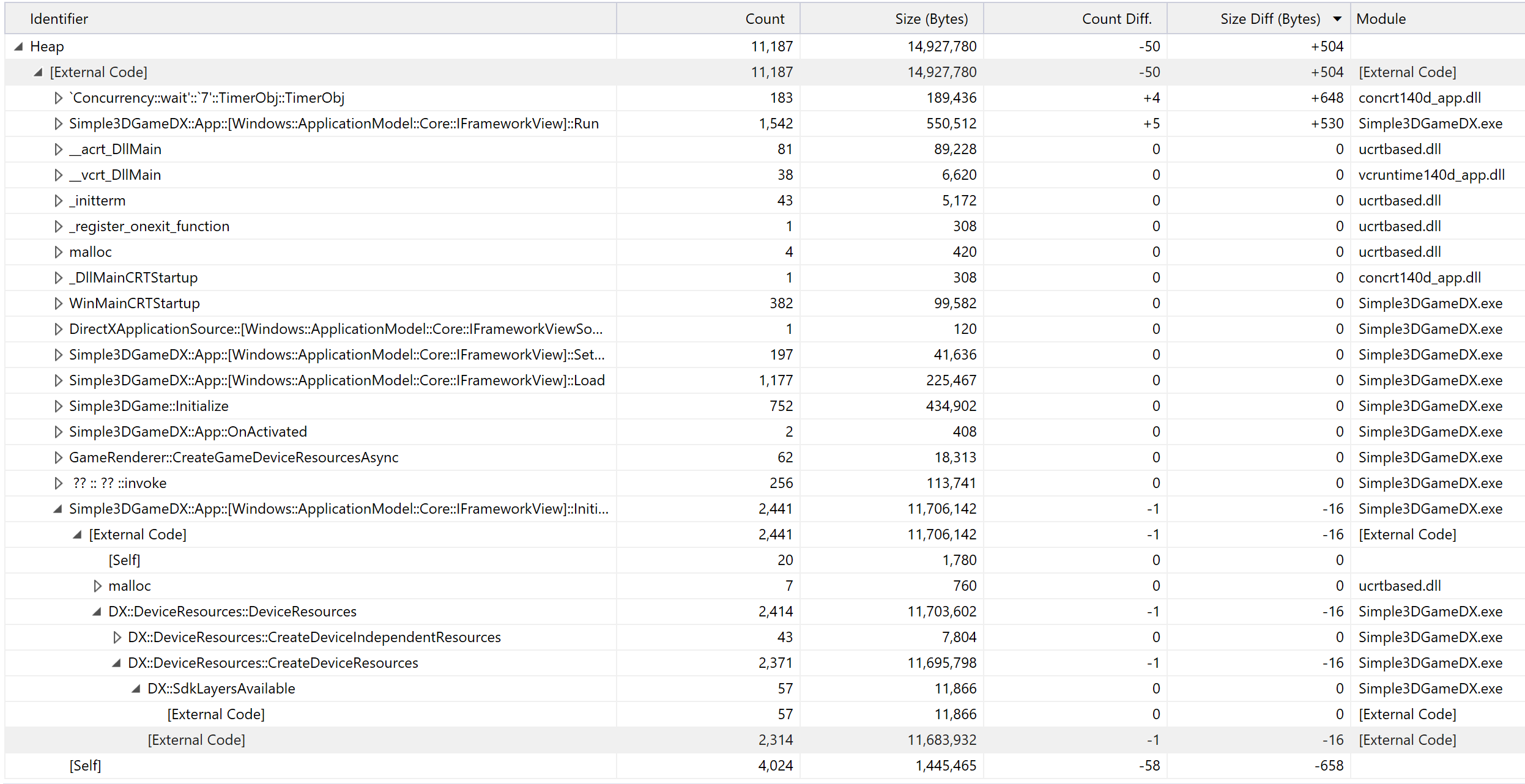
|  |
| --- |
| Only if you want something interesting you need: (warning 2.3 GB)  <https://developer.microsoft.com/en-us/windows/downloads/windows-10-sdk>  and:  <https://github.com/Microsoft/Windows-universal-samples/archive/master.zip>  Specifically, we want the Simple3DgameDx  Extract the zip, click the .sln file for the Simple3DgameDx, and then make sure you can build it first. By default it tries to build for an ARM system, so you need to click the drop down beside debug to say x64 or x86.  Make sure it builds and runs |

To run your program press Alt+ f2 (note that this keyboard shortcut is the same as in older versions of visual studio, in some versions it’s called “Performance Profiler” in others it’s called “Start Diagnostic Tools without debugging”.

Click the box for cpu.



Do it again except for application timeline, GPU and Memory (note that for memory to work you need to click the take snapshot button by tabbing out).



Network is conceptually the same, but the application we’re mucking with probably doesn’t have any network functionality. An example explanation for my sample above, the lowest you can get is that of 14MB of application 11.7 is taken up by “External Code” from the DX:SdkLayersAvailable

Voila now you know how to dig through your program source code to figure out why something is so damn slow.

**Have a look at profiles for CPU, GPU and memory**. You may also want to try this on one of your own programs, but, of course, most programs you guys have written won’t have many methods to do anything when profiled.

**Screenshot the CPU, GPU and Memory profile of one of the sample applications, paste those into a word document and submit on blackboard.**

## Some quick theory discussion

You should probably know something about this for the exam.

This lab then brings us back to lecture 1. How much speedup do you get from a program by speeding up a particular function? Or which is a better measure of performance, response time, throughput etc? And now we can measure those. If you have (as is often the case) one particular method that’s troublesomely slow, or producing wrong results you can look at the assembly output for that one function and see if maybe something is wrong.

That said, there are different ways programs can be profiled, and those choices have consequences to how the profiler behaves and whether or not profiling things is going to change how your program runs.

There are basically 3 schools of thought –

**Event based profilers** (including in .net and java) which basically send an extra message every time a method is called, and that message says something about the state of the method. The upshot here is that it’s easy to gather data, the downside is that you incur a potentially significant increase in messages floating around memory. On a multi-processor environment, you can use this really well, for example if you target your program to run on a 4-core system but run an event profiler on a 6-core system 2 cores can receive and process all the messages.

**Sampling profilers**, interrupt program operation in a random(ish) way, and samples the instructions. The usual problems with sampling come into effect, and you then randomly change how the program executes.

**Instrumentation** rewrites your program by adding in instructions (usually automatically) that record the data you want. This is basically a sophisticated version of adding a “System.out.println()” to every line of code and having it display the current state of the function. This completely screws with how your program actually runs but gives you exact details about what is happening. Useful for when your code is giving you incorrect results rather than slow results.

The 4th option is both obvious and absurd at the same time. That’s where you build custom hardware that is a superset of existing hardware, with, for example, extra memory (every cache, or program counter or whatever has extra memory with information about it’s state). This is absurdly expensive, though both Intel and ARM have some circuitry that does this in regular hardware (which instrumentation software can read).