**Why I Chose the Topic**

This challenge is related to [my research project in BUVIAR Lab](https://github.com/demet47/ML-Data-Collection-via-VR). In the project, I wrote a script to take an RGB and depth image of the scene with a frequency I specify, which is an independent variant. But as I increase the frequency, I realize that the game periodically freezes for a really short but hard not to notice time interval. We always perceive games as fun tools, I am aware that this context is a little far away and a little niche among the ones I could have chosen, but this was a problem I experienced from first hand and I wanted to search and experiment on the possible solutions I would have tried if I wanted to eliminate it.

**Experimental Design**

I wanted to display a basic and simultaneously tractable settings that vary only on a single aspect that I believe would be sufficient to reflect the purpose of the experiment. I chose that significant independent variable to be the speed of the objects. The speeds are decided with trial and error: I wanted to set them with higher speeds but they faced tunneling at early stages of the experiment thus I limited them to speed 90.

The scene basically consists of:

1. Four balls with speeds 30, 60, 90. They do a harmonic motion with constant speed, changing directions at each collision with the invisible walls.
2. A wooden floor just for aesthetic purposes: the balls don’t interact with the floor as they are not affected by the gravity thus not touching the floor.
3. A light source
4. Invisible walls surrounding the experiment setting.
5. Texts attached to balls demonstrating their supposed to be speed values. (they deviate as the game proceeds)

I have three Capture Orchestrator objects that act as dependent variables. I will investigate the game elements and parameters activating a single one among them at each run. They technically are objects without renders that has scripts attached to them to execute the screen capture with differing technics. A more detailed explanation is as follows:

1. **Screen capture with coroutines:**

This uses Unity Coroutines which enable me to spread a computationally large function to multiple update frames. I divided the coroutine into three update loops, giving IO manipulation part solely to a single update along with giving pre-IO and post-IO parts to separate update turns. To provide that no multiple capturing events work in parallel but only work serially, I used a mutex-like structure through a Boolean “locked” which is checked at each update loop to decide on igniting the execution of the coroutine.

1. **Screen capture with asynchronous methods:**

This uses asynchronously running Update loops. Asynchronous parts are written with Task typed functions and are called with *await* keywords inside encapsulating asynchronous functions. Another detail is that these asynchronous parts may run on a separate thread from the main game thread, but in our case since Update method runs on main thread, an asynchronous call inside it will also run on main thread. It still is advantageous because it is a non-blocking call to the function; main thread continues execution of other tasks too which also provides parallelism to some extent. In my code, I separated only the IO operation part into the asynchronous part (detailed reason is after this section). I again utilized mutex-like structure using a Boolean *atomicLock* variable to provide that a capture waits for the previous ones’ termination.

1. **Screen capture with Unity jobs:**

Unity job system do not generate as much garbage as asynchronous approach do, and utilize Unity’s worker threads. When used with Burst compiler, I could have created a smoother game bot this still will demonstrate high optimization. I simultaneously used the mutex logic again too to provide that sequential captures are taken and the sequence of screenshots are in line with chronological order, through the Boolean *done*.

The technical challenge here was that I wanted to fixate the atomic code peace that did the experimental act I was testing. Doing so, I faced some restrictions due to threads: some methods opened up separate threads other than the main thread unlike Coroutines and thread switch prohibited some methods’ execution. But gladly, the significant part that created the greatest bottleneck, the disk IO part, was the part I could run at a separate thread. The screen capture method doesn’t use the easy method UnityEngine provides (ScreenCapture.CaptureScreenShot).

***Below is the screenshots from Profiler views, the timings being the initial interval of the runs:***

*From top to bottom: Job System, Coroutine, Asynchronous*

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

Table 1

**Experiment Results**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Average Screen Capture At Time (seconds) | | | | Approximate Time When | CPU Utilization |  |
| 30 | 60 | 90 | 120 | Visible Mishaps Start (sec) | At the Termination in secs (per frame) |  |
| 12.1 | 12.5 | 11.7 | 11.4 | 71 | S: 66.2 ms R: 10.31 ms P: 2.34 ms | Coroutine Try 1 |
| 11.2 | 12.2 | 12.4 | 12.1 | 160 | S: 60.99 ms R: 10.76 ms P: 1.88 ms | Coroutine Try 2 |
| 10.9 | 12 | 12.3 | 12.34 | 147 | S: 63.68 ms R: 11.44 ms P: 0.54 ms | Coroutine Try 3 |
| 9.58 | 9.3 | 9.1 | 9.15 | 114 | S: 71.21 ms R: 11.34 ms P: 0.92 ms | Coroutine Try 4 |
| 10.8 | 12 | 12.3 | 12.5 | 187 | S: 63.30 ms R: 7.54 ms P: 1.39 ms | Coroutine Try 5 |
| 12.4 | 12.9 | 12.96 | 12.75 | 133 | S: 61.53 ms R: 8.47 ms P: 2.97 ms | Coroutine Try 6 |
| 12.1 | 12.8 | 12.97 | 12.5 | 118 | S: 57.81 ms R: 9.76 ms P: 0.75 ms | Coroutine Try 7 |
| 11.2 | 12.4 | 12.7 | 12.2 | 111 | S: 67.63 ms R: 10.46 ms P: 0.56 ms | Coroutine Try 8 |
| 11.5 | 11.8 | 11.9 | 11.9 | 107 | S: 66.15 ms R: 9.93 ms P: 0.65 ms | Coroutine Try 9 |
| 11.1 | 11.7 | 11.7 | 11.7 | 74 | S: 60.24 ms R: 10.28 ms P: 1.02 ms | Coroutine Try 10 |
| 11.288 | 11.96 | 12.003 | 11.854 | 122.2 | S: 63.87 ms R: 10.03 ms P: 1.302 ms | AVERAGE |
| 15.7 | 16.5 | 16.7 | 16.8 | 177 | S: 61.07 ms R: 9.14 ms P: 1.00 ms | Async Try 1 |
| 14.1 | 15.1 | 15.2 | 14.5 | 148 | S: 58.15 ms R: 8.66 ms P: 0.00 ms | Async Try 2 |
| 12.7 | 13.6 | 13.6 | 13 | 78 | S: 61.71 ms R: 9.10 ms P: 0.26 ms | Aysnc Try 3 |
| 14.9 | 15.5 | 15.5 | 14.9 | 147 | S: 60.16 ms R: 56.74 ms P: 0.77 ms | Async Try 4 |
| 14 | 15 | 15.5 | 15.58 | 111 | S: 50.08 ms R: 9.42 ms P: 1.23 ms | Async Try 5 |
| 14.8 | 15.7 | 15.8 | 15.4 | 116 | S: 85.25 ms R: 53.53 ms P: 0.32 ms | Async Try 6 |
| 11.7 | 12.3 | 12.4 | 12.37 | 86 | S: 57.95 ms R: 9.41 ms P: 0.3 ms | Async Try 7 |
| 14.6 | 14.8 | 14.6 | 14.3 | 91 | S: 80.72 ms R: 9.23 ms P: 0.0 ms | Aysnc Try 8 |
| 10.3 | 11.4 | 11.7 | 11.9 | 86 | S: 61.87 ms R: 9.45 ms P: 0.05 ms | Async Try 9 |
| 12.9 | 13.48 | 13.7 | 13.06 | 97 | S: 56.71 ms R: 9.29 ms P: 0.26 ms | Async Try 10 |
| 13.57 | 14.338 | 14.47 | 14.181 | 113.7 | S: 63.37 ms R: 18.39 ms P: 0.419 ms | AVERAGE |
| 15.2 | 15 | 14.5 | none | 80 | S: 54.06 ms R: 8.72 ms P: 0.75 ms | Job Try 1 |
| 14 | 14.5 | 14.6 | none | 98 | S: 72.29 ms R: 34.07 ms P: 2.6 ms | Job Try 2 |
| 14.5 | 15.6 | 15.7 | none | 77 | S: 60.6 ms R: 9.35 ms P: 0.92 ms | Job Try 3 |
| 13 | 14 | 14.2 | none | 100 | S: 70.88 ms R: 47.21 ms P: 1.1 ms | Job Try 4 |
| 13.4 | 14 | 14.2 | none | 100 | S: 49.43 ms R: 8.24 ms P: 0.42 ms | Job Try 5 |
| 13.6 | 14.3 | 14.5 | none | 87 | S: 64.12 ms R:10.81 ms P: 1.02 ms | Job Try 6 |
| 15.5 | 16.1 | 16.2 | none | 54 | S: 58.15 ms R: 9.95 ms P: 0.92 ms | Job Try 7 |
| 11.9 | 12.9 | 12.8 | none | 69 | S: 69.6 ms R: 9.56 ms P: 0.6 ms | Job Try 8 |
| 13.2 | 13.9 | 13.8 | none | 88 | S: 71.29 ms R: 9.44 ms P: 1.02 ms | Job Try 9 |
| 13.5 | 13.9 | 13.9 | none | 87 | S: 69.6 ms R: 9.56 ms P: 0.6 ms | Job Try 10 |
| 13.78 | 14.42 | 14.44 | none | 84 | S: 64.0 ms R: 15.69 ms P: 0.99ms | AVERAGE |

Table 2

**About the Metrics I Selected**

The bottleneck in my scenario is created due to frequent IO operations. It can be cleverly positioned and handled such that it isn’t blocking other operations, keeping them busy-waiting. The blocking behaviour can be observed from the trend it follows in Profiler visually. In Table 1 in previous section, we have an initial view to them, which we will comment on later. Other than that, I recorded average screen captures per second to have an idea about the efficiency of the methods selected: I don’t set a certain frequency, each capture is initiated right after the termination of previous one with the help of a mutex mechanism, ignition being inside Update functions. I also observed visually the smoothness of the rendering, more specifically the game experience itself visually. As the captures push the limit of my permanent memory’s limitations, the Profiler gets denser, and at the end the game abruptly stops by giving the error that we basically are out of memory. I chose a limit time to proceed the game and recorded the values I see on profiler for CPU utilizations per task (**S** for Scripts, **R** for Rendering, **P** for Physics).

A detail here: I wanted to fixate this value in 200. But per approach, the termination of game changed and not all of them lasted 200 seconds. So the duration of each experiment trial is as follows per approach:

* 200 seconds 🡪 Coroutines
* 140 seconds 🡪 Asynchronous
* 90 seconds 🡪 Job system

**Comprehension Of The Collected Data**

Starting from the visual trends I observed in Profiler GUI and the screenshots in Table 1, we can clearly see that the most parallelization is done in Job System. It is apparent there is a period behaviour for each CPU utilization in Coroutines and Asynchronous Approach where some processes are being blocked (pits of the waves). This is reduced in asynchronous case; the wave is more dense but completely eliminated in Job System. The overall CPU utilization increased for all of them as the game proceeded. But the most homogeneous trend was observed in asynchronous case. Job System had some glitches with high peeks although not observable in rendered game view. ***All of them after a threshold time approached 66ms frame time, initially async and coroutine case being closer to 33ms frame time.***

From Table 2, we see a similar ordering of performance:

Job System > Asynchronous Approach > Coroutines

This is not a surprise. Coroutines don’t eliminate the blocking behaviour of IO, it just isolates this operation from other computations, relieving it a little bit.

Asynchronous Approach inhibits the busy waiting cycle. It provides a non-bocking wait. We are informed of the termination of IO operation via the status of mutex.

As for the Jobs, we directly do the IO operation on separate worker threads. Main thread continues with the game flow simultaneously.

The average screen capture per second values are highest in Job Systems, then in Asynchronous, the worst being Coroutines as expected below. The second metric, approximate time of visible mishap occurrence has a nuance: For Coroutines, the visual discreetness was visible right from the beginning for Coroutines. I recorded the times when it got worse. For asynchronous, it was smoother but the render was again consistently discrete after a threshold time. I cannot say this for Job System; for jobs, the render experience never was disturbing. I really put effort to observe a discrepancy, lag or discreet velocity but in some trials I couldn’t even catch a single frame like that. Also, the abrupt termination got closer, so the timing of the initial visible mishaps also may be related to this fact. The numeric values for the CPU usage at the termination doesn’t tell me much at first sight. They are close to each other. The abrupt termination was earliest in the fastest and smoothest method, latest in the least smooth and slowest method (former being Job system and latter being Coroutines).