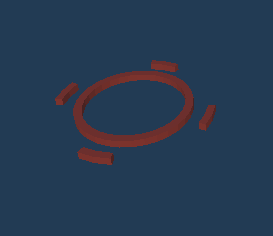
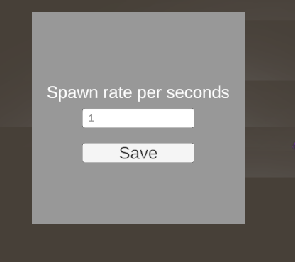
**Read Me Exploding Elves prompt.**

**Instructions**

This is a game where the player needs to start a new match for the elves sparring spawning. In the game, there are 4 types of elves (black, red, white, blue). They spawn through the warp or spawn point.



**[Warp/Spawn point]**

The player can change the spawn point rate by clicking on it at any time during the app's run. By clicking on the warp, a pop-up will appear, allowing the player to change the spawn rate by inserting a number; this number can be a decimal. 

**[Warp Set Rate Popup]**

When the player presses the "Star Matcht" button, elves are spawned through the warp at the rate set by the player. If the player doesn't set a rate, the default rate is one elf per second.

The elves have three behaviors:

* The elves should move randomly in the scene. And can´t be controlled by the player
* If elves with the same color collide, they should generate one extra elf of the same

type. A little blip sound and a purple VFX plays every time an elf is spawned

* If different colored elves collide, both should explode. A explosion sound and an explosion VFX should play when elves explodes

When the match starts, the "Start Matxh" button is replaced by an "End Match" button, which can be pressed by the player at any time during the match. And, will disable all elves in the game.

**Used Concepts**

I tried to use SOLID principles where it made sense. Even in this simple and small project, I separated responsibilities into different classes and had a class to manage them using the Single Responsibility Principle.

Moreover, I tried to avoid coupling classes by using Actions and the Observer Pattern.

I ensured that every SpawnPoint has a single pool, as well as the VFX, to avoid garbage collection and control memory usage. The game developer can set the maximum number of objects that can be in the pool through the Unity inspector in the project.

**Improviments:**

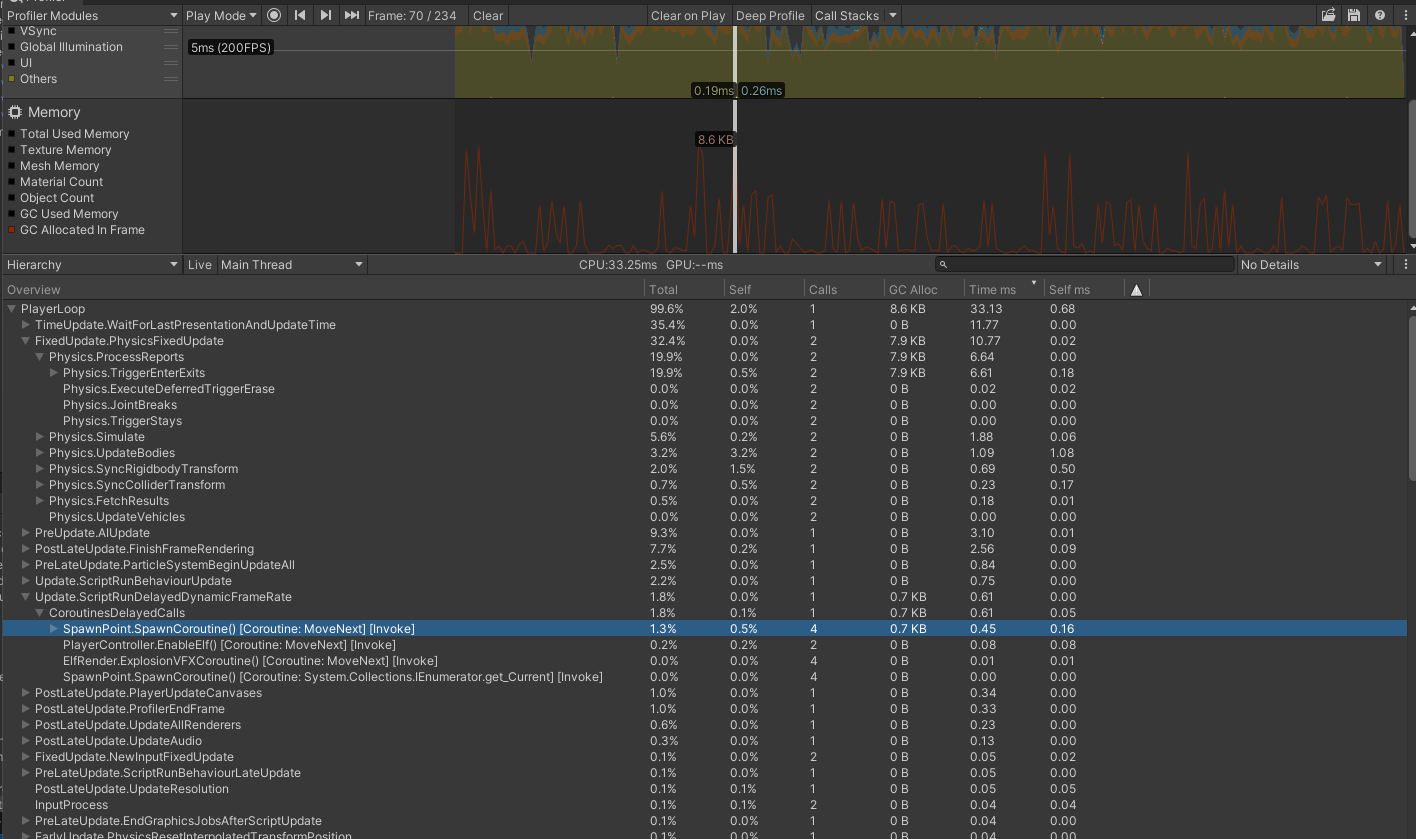
* Create a generic pool to avoid duplicating code.
* Enhance UI with tweens and better icons.
* Create a splash screen or a menu screen to better organize the game.
* Consider creating a single pool for all elves, as generally only two types of elves have the majority of life, and the others still occupy memory even when they are not frequently seen in the scene.

**Report on memory usage.**

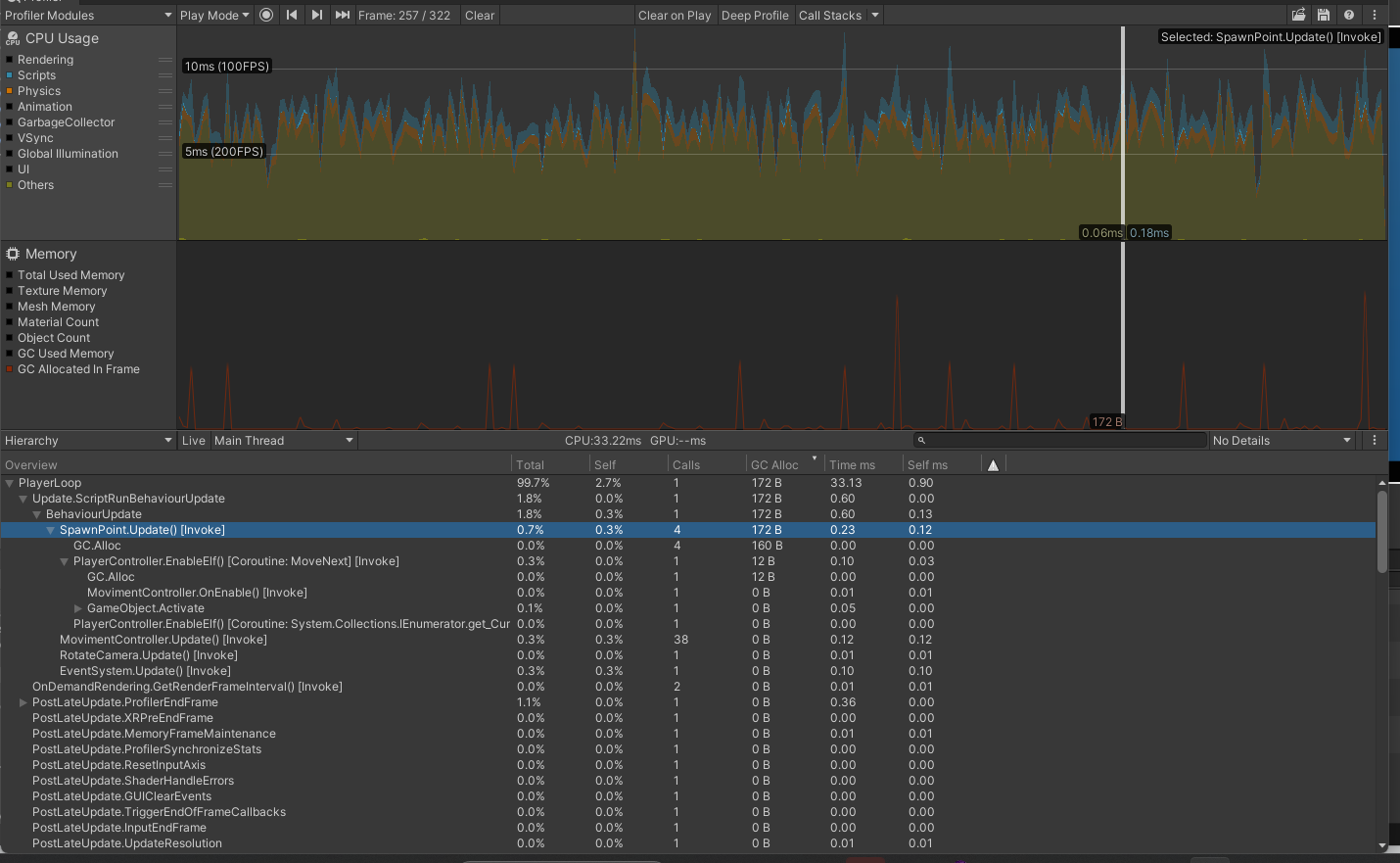
**Garbage Collector:**

The VFXs and the Elves were in pools to avoid the need for garbage collection. However, upon inspecting the Profiler, it was observed that Coroutines are geratating garbage. To address this, I consider what was better to put on the MonoBehaviour.Update and what was better to continue using coroutines.

**Spawn Elves:**

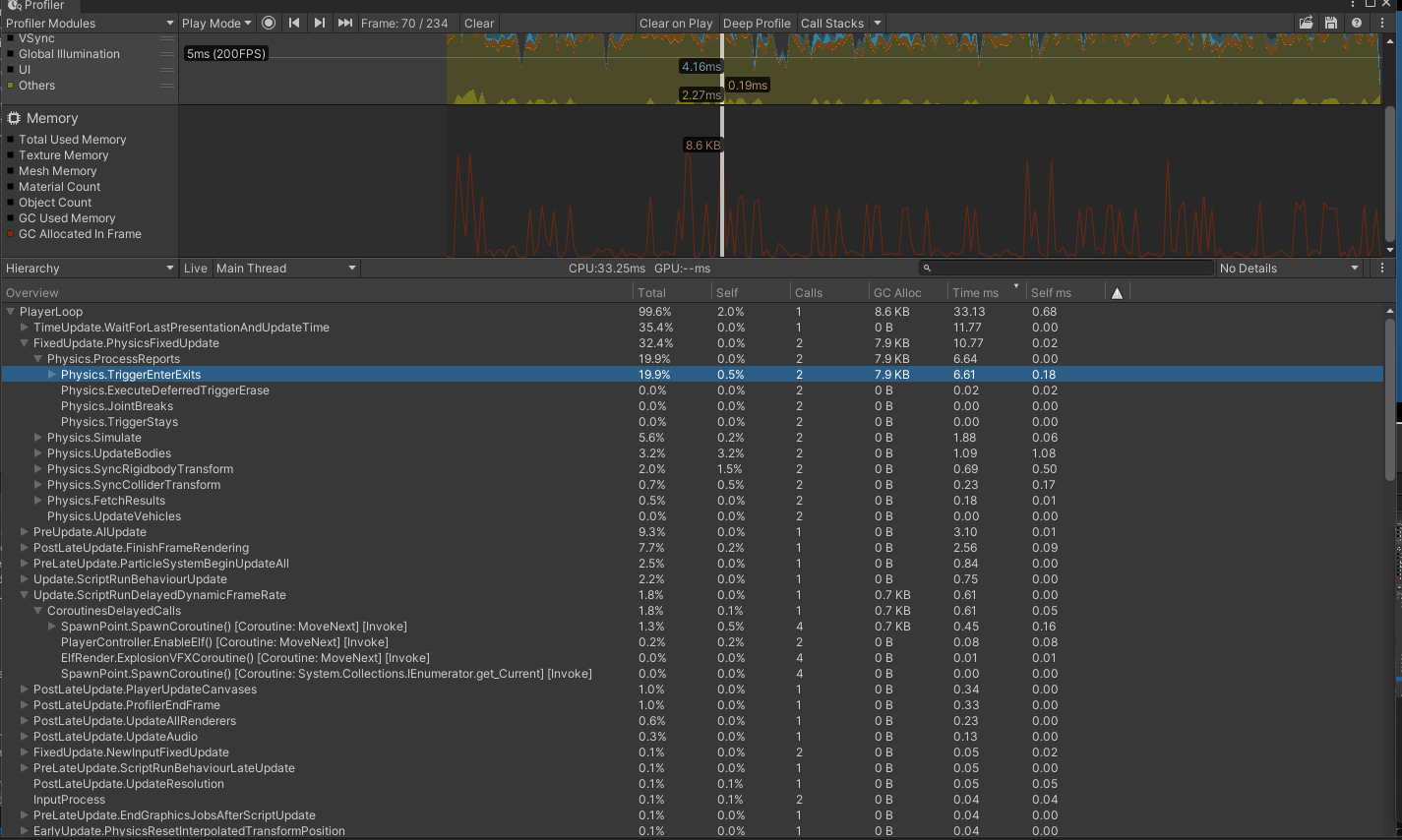
The function responsible for spawning elves was replaced in the update loop because it is the main behavior of the game and is called 4 or more times than other functions. Considering the frequency, it is better to place it in the update loop.****

**[GB profiling With spawn rate using coroutine]**



**[GB profiling with Spawn Elves controlled on MonoBehaviour.Update]**

**PhysPhysics.TriggerEnterExi**t is another function that calls the garbage collecto frequently. However, for this project, I don't see significant drawbacks in changing the use of this function with private code.

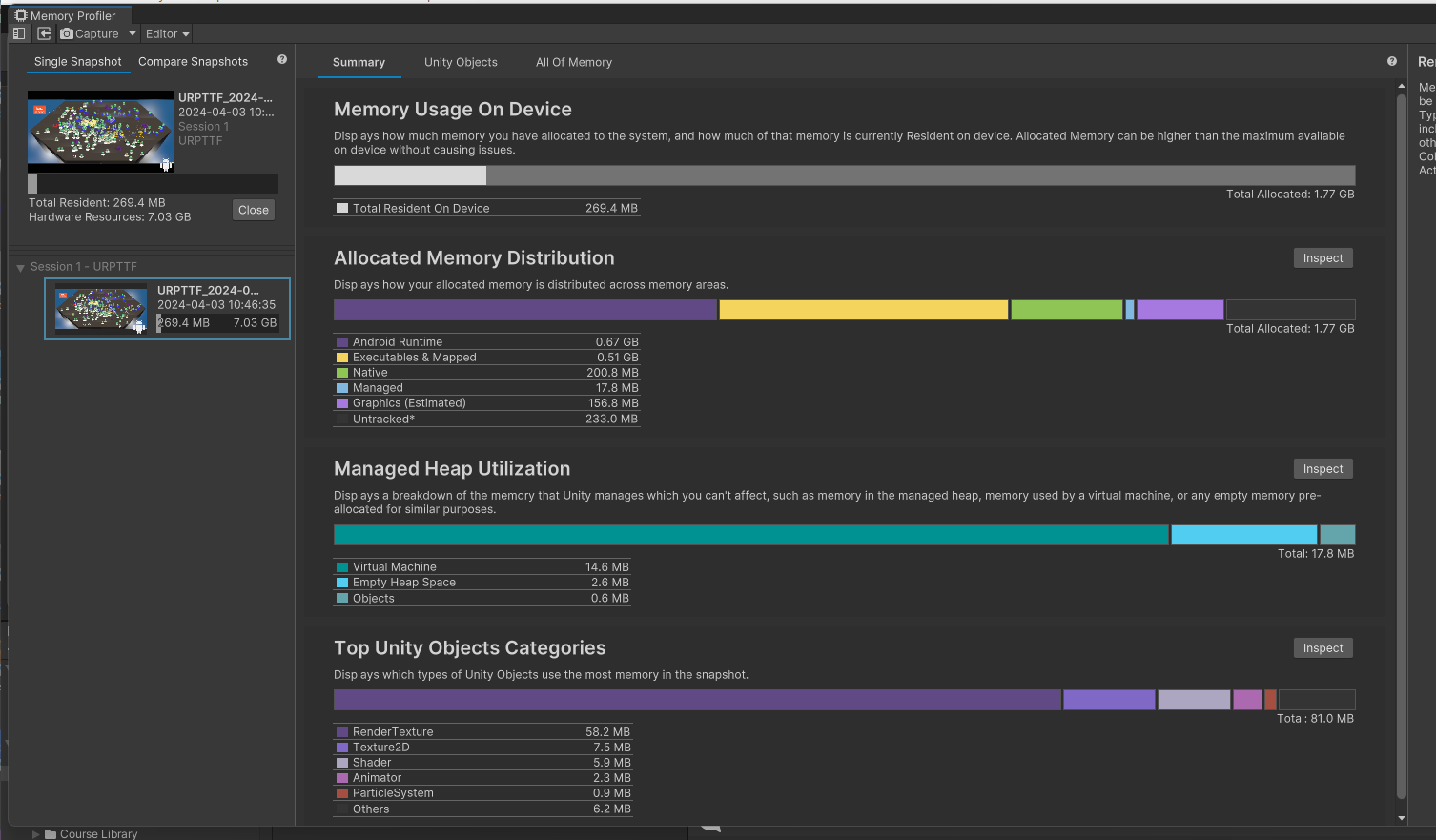


**Notes:**

I can only test on an Android device since that's all I have access to.

**Memory Profiler:**

The captures were made on the Android target.



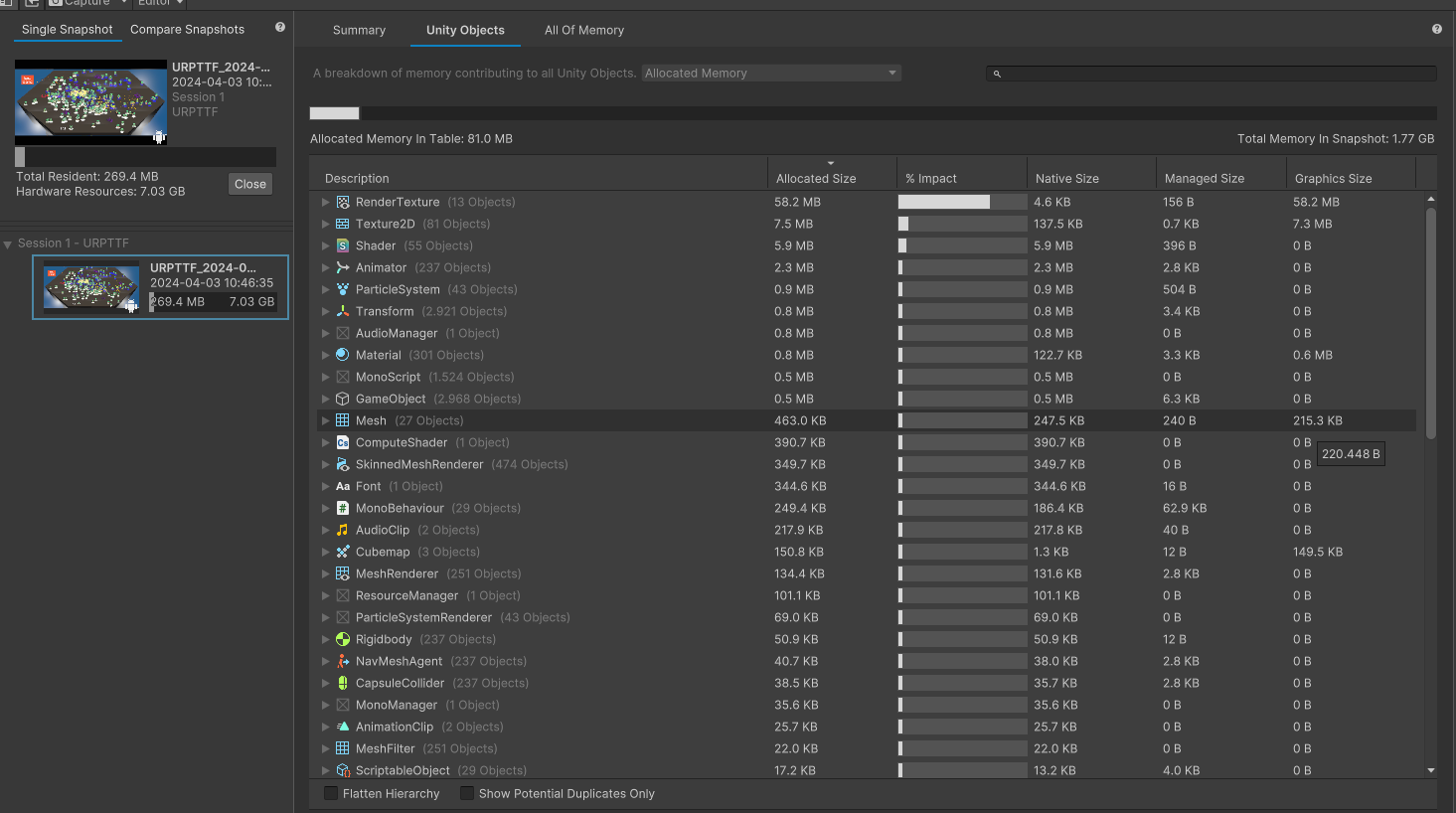
**Allocated Memory Distributions:**

The **Native** category is taking up 200.8MB, showing allocations of objects such as SceneObject, Native Allocations, Assets, and Manager.

Memory Manager is responsible basically for the operation of the garbage collection. Considering that it is a small project and I pooled objects to avoid as much as possible garbage collection, I believe the number of 17.8MB is low.

**Managed Heap Utilization:**

It breaks down the Managed category from Allocated Memory distribution, showing the memory used on the C# scripts and also the managed garbage collection.



**Unity Objects:**

Unity objects are consuming only 81.0MB of memory. RendererTexture is the most expensive in terms of memory usage. Assets were obtained from different sources, so a possible optimization is to create a Unity sprite atlas or optimize them outside Unity.

**Notes:**

* I can only test on an Android device since that's all I have access to
* Adicional Unity packages
  + Ai Navigation
  + Memory profiler
  + TextMeshPro