Kalvin Malloch
1902581

# COMP280 Worksheet 3: Optimisation

# Contents

# 1 Introduction

## 1.1 Repository

COMP280 Worksheet 3 Repository (Includes Optimized changes):
*https://github.com/KalvinMalloch/comp280-worksheet-3*

Creativity Cards Project Repository (No Optimized changes):
*https://github.com/DanielNeale/CreativityCardsGroup6*

## 1.2 Chosen Game


Figure 1. The main menu for 'Keep Dogs on a Lead'.

I chose a project that I was involved with in first year called 'Keep Dogs on a Lead' which was for our Creativity Cards project. It is a game consisting of a man and a dog who are on a beach metal detecting and selling anything they find. It was made in about two weeks so the game did not have much polish to it as we could not focus on maintainability. Therefore, I thought it was a perfect project to try and optimize.

# 2 Small Tweaks

## 2.1 Water Vertices

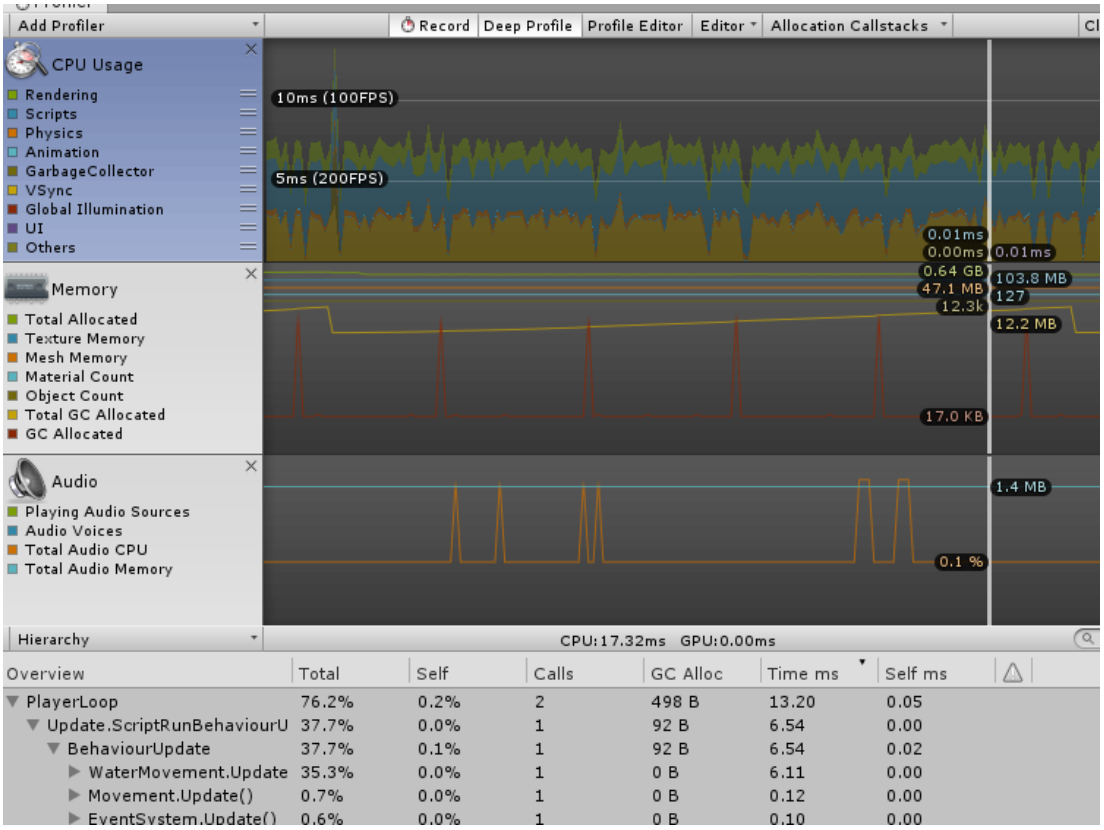



Figure 2. The first recorded benchmark for the game.

As Figure 2 shows, this was the first benchmark I recorded when I played the game, running at an average of 150 FPS without moving the player or anything in-game. The first point I noticed was that 35% of the total usage was being used by a 'WaterMovement' class which only handled the moving of the ocean waves in the background. As shown in Figure 3, the waves are generated with vertices to simulate a moving ocean.

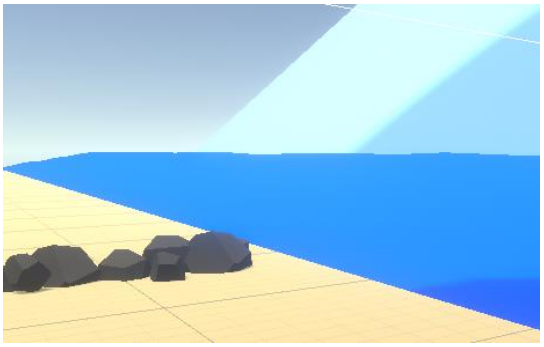


Figure 3. The ocean wave vertices changing at random.

Using the hierarchy, I followed 'WaterMovement.Update()' down to the function that was causing the usage to heighten. After finding it, I opened Visual Studio to see how I could optimize this issue.

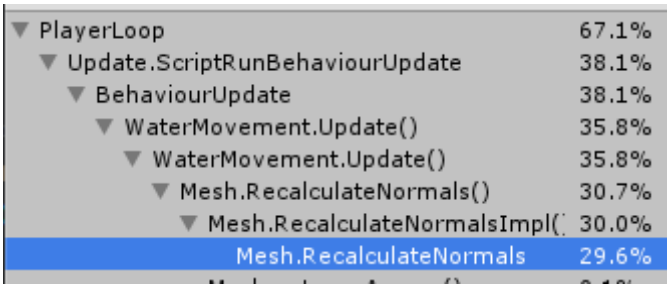



Figure 4. The hierarchy showing where most of the CPU usage is coming from.

The hierarchy showed me that calling 'RecalculateNormals' was the issue and after opening the script, I could see why. It was being called inside a for loop for every vertex present in the mesh as seen in Figure 5. This probably meant that it was recalculating the normals of the mesh for every vertex that was present in it – which is more often than it needed.

```
void Update()
{
    for (int i = 0; i < verticies.Length; i++)
    {

        if (targetHeight[i] > verticies[i].y)
        {
            verticies[i].y += waveSpeed;

            if (targetHeight[i] < verticies[i].y)
            {
                targetHeight[i] = Random.Range(-
            }
        }

        else if (targetHeight[i] < verticies[i].
        {
            verticies[i].y -= waveSpeed;

            if (targetHeight[i] > verticies[i].y)
            {
                targetHeight[i] = Random.Range(-
            }
        }
        water.vertices = verticies;
        water.RecalculateNormals();
    }
}
```

Figure 5. The function that handles the water movement and vertices.

My solution for this was simple. If I brought it out of the for loop, I could just have it be recalculated through the 'Update' function instead, saving it from being called upon rapidly and raising the CPU usage. The resulting code change is shown in Figure 6.

```
        }
        water.vertices = verticies;
    }
    water.RecalculateNormals();
}
```

Figure 6. The normals being recalculated outside of the for loop.

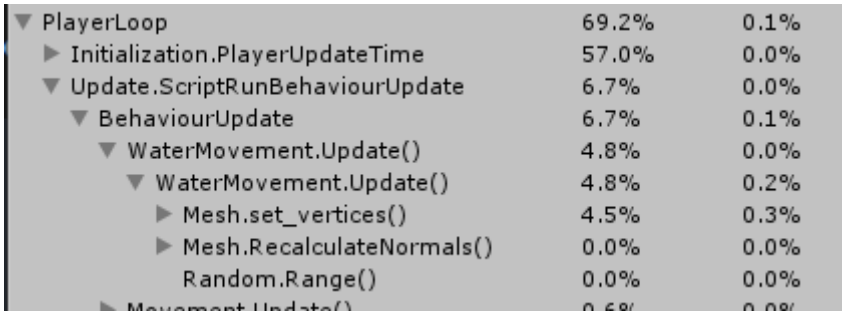



Figure 7. The hierarchy showing the usage of 'WaterMovement' after the change.

As you can see in Figure 7, the change lowered the 'RecalculateNormals' usage from 29% to 0%, successfully optimizing that area.

## 2.2 Printing & Debug Log

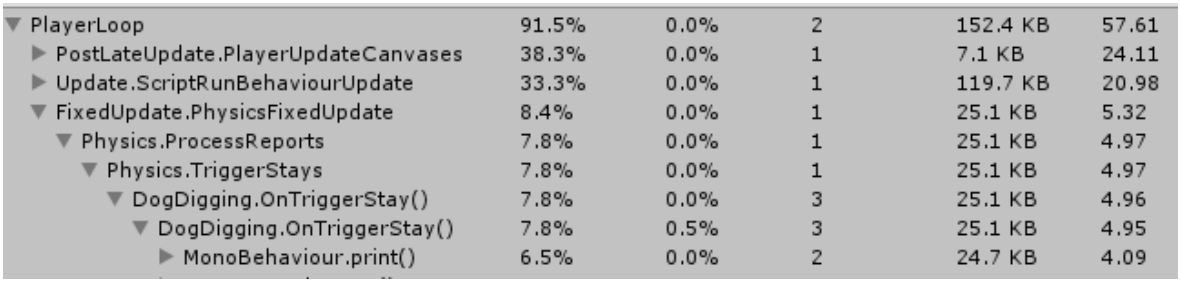
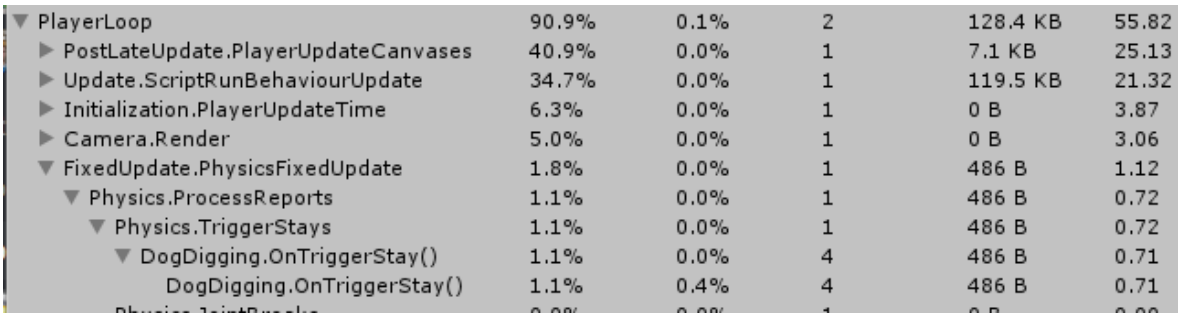



Figure 8. The hierarchy showing the usage of printing.

I noticed that printing text into the console increased usage for a few frames as sometimes they were being constantly told to print, as shown in Figure 8. These commands are usually removed from the final build to improve performance and the size of the build since it can continue logging information. Though minor, I decided to go through the scripts being used and remove any log debugging since this was our final build of our game.

```
void OnTriggerStay(Collider other)
{
    if ((other.gameObject.tag == "Dig
    {
        var item = other.GetComponent
        if (item)
        {
            inventory.AddItem(item.it
            Destroy(other.gameObject)
        }
        print("Got Treasure!");
        Destroy(other.gameObject);
        movementScript.dogDug();
    }
}
```

Figure 9. An example of text being printed for debugging purposes.



| | | | | | |
|---|---|---|---|---|---|
| PlayerLoop | 90.9% | 0.1% | 2 | 128.4 KB | 55.82 |
| PostLateUpdate.PlayerUpdateCanvases | 40.9% | 0.0% | 1 | 7.1 KB | 25.13 |
| Update.ScriptRunBehaviourUpdate | 34.7% | 0.0% | 1 | 119.5 KB | 21.32 |
| Initialization.PlayerUpdateTime | 6.3% | 0.0% | 1 | 0 B | 3.87 |
| Camera.Render | 5.0% | 0.0% | 1 | 0 B | 3.06 |
| FixedUpdate.PhysicsFixedUpdate | 1.8% | 0.0% | 1 | 486 B | 1.12 |
| Physics.ProcessReports | 1.1% | 0.0% | 1 | 486 B | 0.72 |
| Physics.TriggerStays | 1.1% | 0.0% | 1 | 486 B | 0.72 |
| DogDigging.OnTriggerStay() | 1.1% | 0.0% | 4 | 486 B | 0.71 |
| DogDigging.OnTriggerStay() | 1.1% | 0.4% | 4 | 486 B | 0.71 |

Figure 10. The hierarchy showing the change in CPU usage after removing anything for debugging purposes.

As you can see in Figure 10, the change lowered the 'OnTriggerStay' for 'DogDigging' usage from 7.8% to 1.1%, successfully optimizing that area.

## 2.3 V-Sync

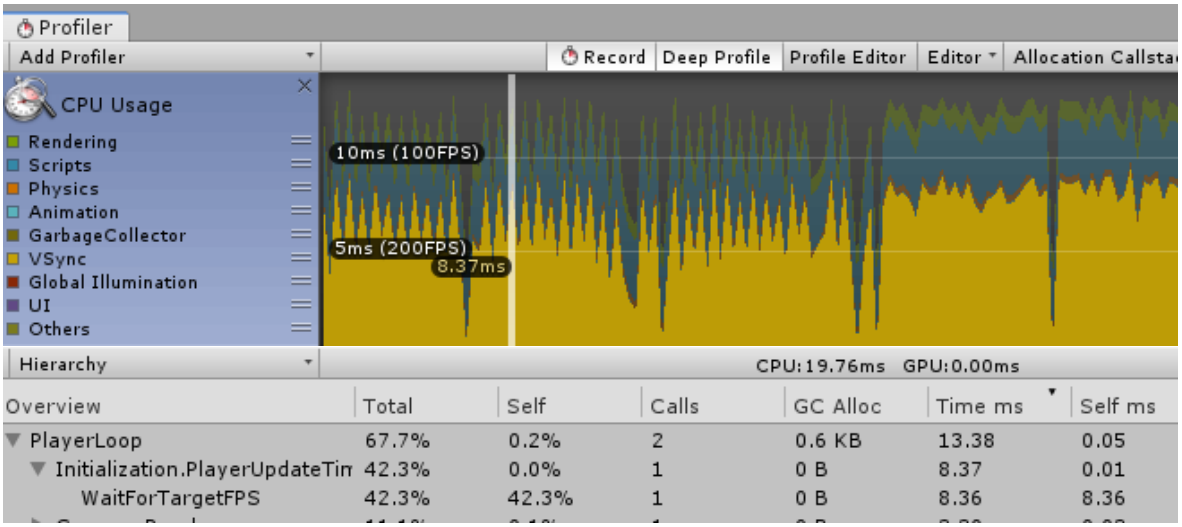



Figure 11. The hierarchy showing the change in CPU usage after removing anything for debugging purposes.

Something I was interested in was the 'WaitForTargetFPS' that was using 42% of the CPU usage, as shown in Figure 11. I was aware that you could have a value to be the target frame rate, but I could not find it declared in any of the scripts. Instead, it is being set by V-Sync as it is syncing the framerate to the monitors refresh rate.

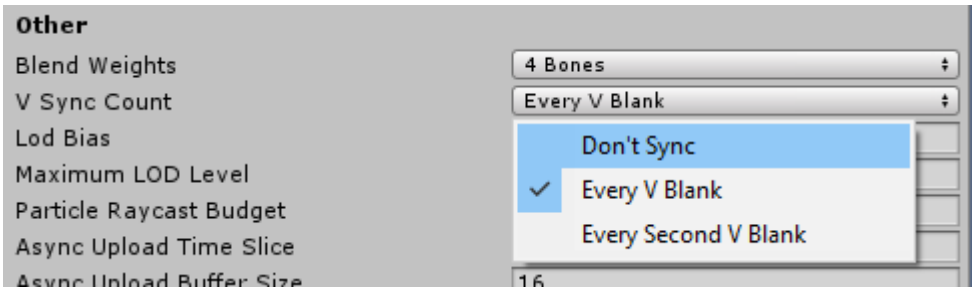



Figure 12. The options for v-sync.

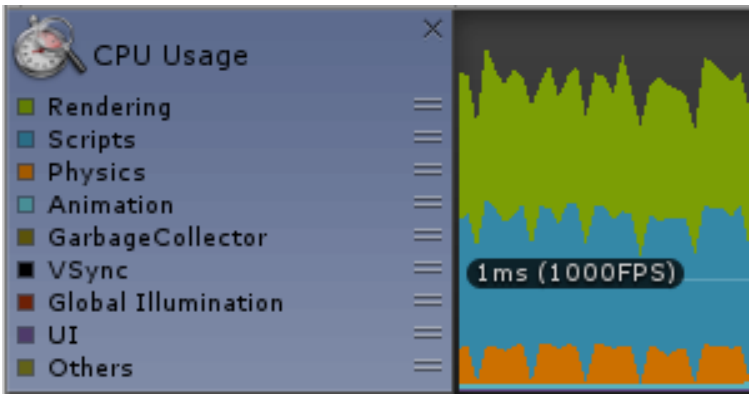



Figure 13. The toggle box that can show you the CPU usage with v-sync disabled.

My first thought was to disable v-sync using the method in Figure 12, but upon inspecting the profiler, you can toggle v-sync off while the profiler updates the CPU usage, shown in Figure 13. This was useful in seeing what FPS the project could run without limitations, hitting around 300 FPS on average.

# 3 Large Tweaks

## 3.1 Canvases

| Overview | Total | Self | Calls | GC Alloc | Time ms | Self ms |
|---|---|---|---|---|---|---|
| ▼ PlayerLoop | 86.3% | 0.2% | 2 | 165.7 KB | 24.42 | 0.06 |
| ▼ PostLateUpdate.PlayerUpdate | 26.5% | 0.0% | 1 | 83.6 KB | 7.52 | 0.00 |
| ▼ UIEvents.WillRenderCanva | 26.5% | 0.0% | 1 | 83.6 KB | 7.52 | 0.00 |
| ▼ UGUI.Rendering.UpdateI | 26.5% | 0.1% | 1 | 83.6 KB | 7.52 | 0.03 |
| ▼ Canvas.SendWillRend | 26.4% | 0.0% | 1 | 83.6 KB | 7.47 | 0.00 |
| ▼ Canvas.SendWillRe | 26.4% | 0.0% | 1 | 83.6 KB | 7.47 | 0.00 |
| ▼ CanvasUpdateRe | 26.4% | 0.0% | 1 | 83.6 KB | 7.47 | 0.00 |
| ▼ Layout | 26.4% | 0.1% | 1 | 83.6 KB | 7.47 | 0.04 |
| ▶ Graphic.Reb | 25.9% | 0.2% | 30 | 83.6 KB | 7.35 | 0.07 |

Figure 14. The hierarchy of the profiler showing a spike in CPU usage related to canvases.

A spike I noticed in CPU usage happens when the player goes over to one of the shops located on the level. When the player walks in front of a store, a canvas game object is set to active and appears for the player to interact with. It is then disabled when the player leaves the area. The spike happens when the player triggers the canvas game object to become active, and specifically the usage is affected by 'Graphic.Rebuild'. This builds the graphic geometry of the canvas but since I am disabling the game object, the canvas will need to rebuild itself every time it is enabled.

To fix this spike in CPU usage, I changed my script so that instead of deactivating the game object as a whole, I just grabbed the 'Canvas' component from the canvas and disabled it when it was not needed. This meant that when the game started, the canvas was built from the beginning and did not need to be rebuilt every time I activated the game object.

```
public void LookAtShop()
{
    vCam.SetActive(true);
    vCam2.SetActive(false);
    shopkeeperAllMenus.SetActive(false);
    goToShopMenu.SetActive(true);
```

Figure 15. The canvases game object being disabled and enabled.

```
0 references
public void LookAtShop()
{
    vCam.SetActive(true);
    vCam2.SetActive(false);
    shopkeeperAllMenus.GetComponent<Canvas>().enabled = true;
    goToShopMenu.GetComponent<Canvas>().enabled = false;
```

Figure 16. The canvases component being disabled and enabled.

As you can see in Figure 17, the rendering for canvas used to spike up to 26.4% of CPU usage but has now been completely removed as the canvas does not ever need to rebuild during the game's runtime. This actually will help me in future game projects to optimize my game before I run into these kinds of issues.

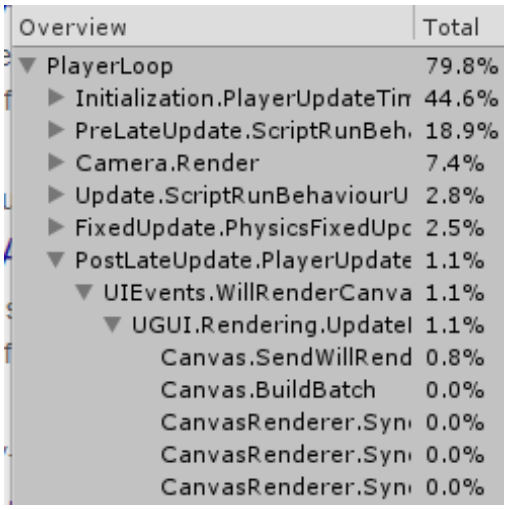



Figure 17. The hierarchy showing the change in CPU usage after stopping the canvas rebuilding.

## 3.2 Instantiating

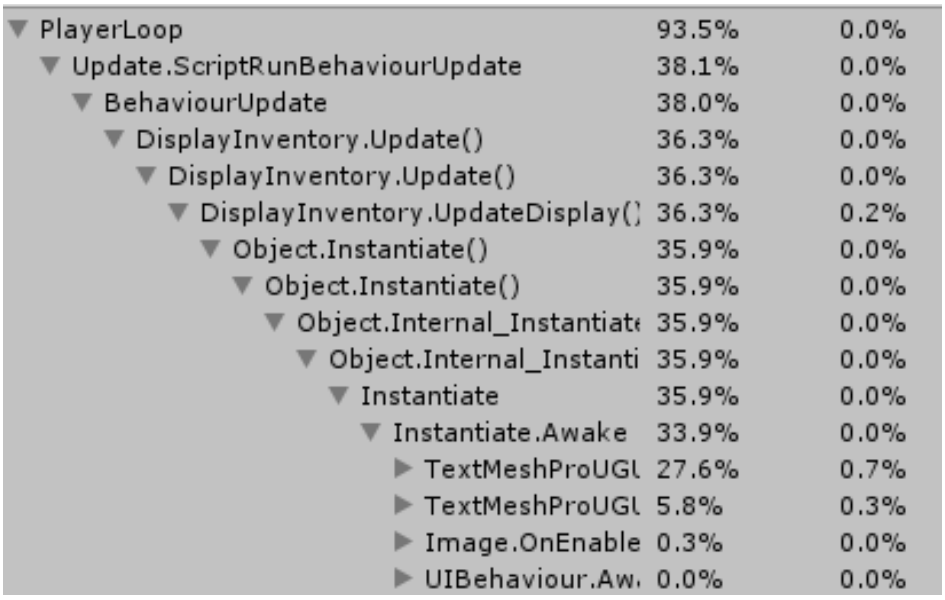



Figure 17. The hierarchy showing the change in CPU usage after stopping the canvas rebuilding.

One of the final mechanics I tested was when the dog digs up treasure, it adds the treasure to an inventory. Upon doing so, there is a usage spike that occurs related to the instantiation of the treasure, as seen in Figure 17.

```
var obj = Instantiate(inventory.Container[i].item.prefab, Vector3.zero, Quaternion.identity, transform);
obj.GetComponent<RectTransform>().localPosition = GetPosition(i);
itemsDisplayed.Add(inventory.Container[i], obj);
```

Figure 18. The code that instantiates the treasure that is dug up by the dog.

After reviewing the code in Figure 18, I was unable to make changes that worked as one of my teammates wrote this, and a script related to storing the treasure. However, to fix the usage spike I would have to use something called an object pool which is like instantiation but less burdening on the CPU. Instead of spawning and deleting instantiated objects, object pooling happens at the start of the game where all the instantiated objects are disabled, but spawned in ready to use. The objects then are enabled when the player is needing use of then, avoiding anything being instantiated during runtime.

# Conclusion

In conclusion, this optimization task has helped me realize what to look out for in current and future projects. Issues like UI rendering and instantiations heighten the CPU usage and can now be resolved using the information I have learnt in this worksheet. I have also learnt how to effectively use the Unity profiler to benchmark my game and find any issues with CPU usage, using the hierarchy to track down where the specific issue lies.