COMP280 Assignment 1
Worksheet 3 - Optimisation

Optimising a personal project (game jam)

Student No. 1904827 - 04/01/2021

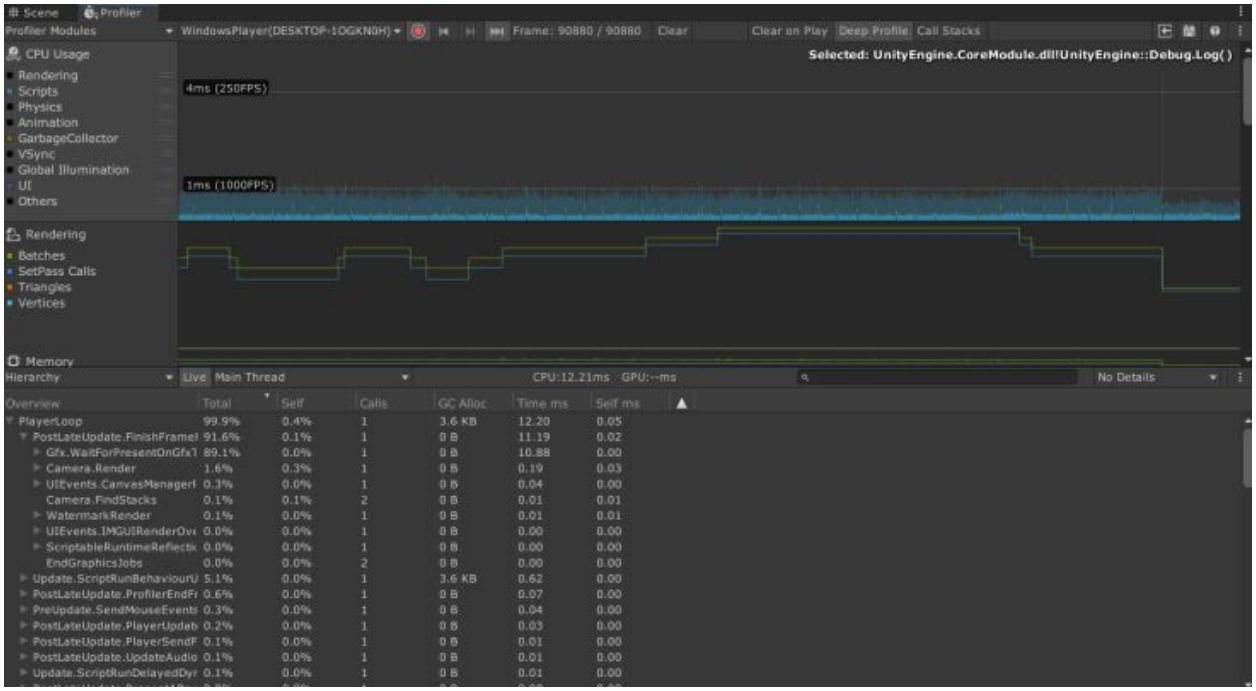
Table of Contents

Optimisation Report

I will be completing my optimisation for this assignment worksheet by taking a look at a personal project of mine from the global game jam in 2019. The project is available here: https://github.com/ColeGGilbert/GlobalGameJam2019

I will be using the unity profiler to identify bottlenecks in the game and/or performance issues caused by UI and/or scripts that I have created. I will provide evidence of these issues via screenshots of the profiler and then I am going to explain where the issue is and how I resolved the problem.

Firstly, I recorded a baseline for the project by running the main gameplay loop and completing two levels before stopping the game and screenshotting the profiler, this screenshot is visible below, showing a version of the project before optimisation:



1 - Identifying problems within the project

After observing the baseline for the project and taking into consideration that this is a smaller project, there isn't a whole lot that is obviously causing issues and that would need to be replaced immediately. However, even with minimal impact, it can be clear which scripts might produce bottlenecks later down the line in the project's development.

The main gameplay loop keeps below the 1ms (1000fps) line, however there is a large spike every so often in the game that reaches above ~4-5ms, which is a large jump compared to the rest of the normal game loop, see Fig 1. I found that this spike is caused by the scene transition between the level and the level completion screen. If I want to keep the scene transition in the game, this spike would be unavoidable, however, it is possible to achieve the same result as what I am aiming for with a simple canvas displaying over the top of the level and then resetting the positions of game objects within the scene. It is beneficial for me to attempt to remove this performance hit since each level can be completed rather quickly, resulting in a lot of scene transitions. As well as this, the scene is reloaded every time the player fails too, which happens more often as the game progresses.

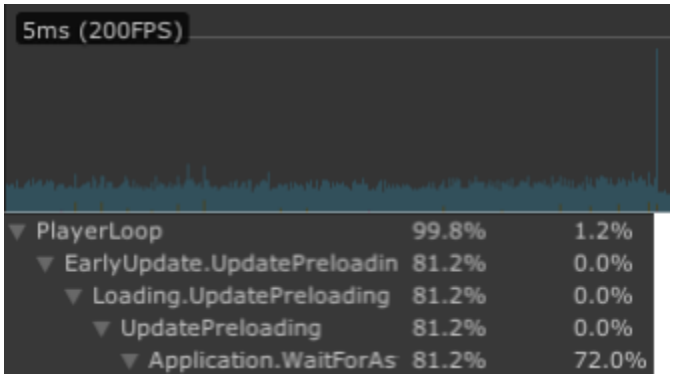


Figure 1. Scene loading creating performance spikes

Other than this large spike, there are some considerable spikes in the main gameplay loop when the game attempts to output a message to the Console through Debug.Log(), see Fig 2. This is unnecessary and can be removed, if not tidied up to avoid causing as much of an impact.

Once these smaller issues are tidied up, the profiler will be easier to read for the main gameplay loop and any larger spikes that appear.

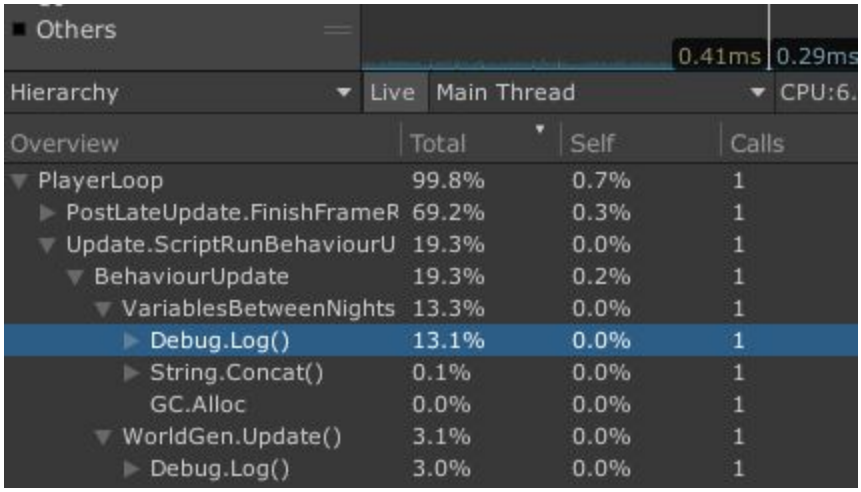


Figure 2. Debug.Log() causing minor performance spikes

1.1 - Resolving the issues that were outlined

Upon further investigation, I found that the events that cause spikes in performance, in relation to running Debug.Log() functions, were most likely causing issues due to each of these being run every single frame of the game, even when displaying no new information. Therefore, to reduce the amount of times that these instances of Debug.Log() are called, I updated my codebase to call the functions only when the information being output is updated.

Before optimisation:

```
void Update () {
    Debug.Log("Wood Remaining: "+woodRemaining);
```

After optimisation:

```
public class WoodIncrease : MonoBehaviour {

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if(collision.gameObject.tag == "Player")
        {
            WorldGen.woodRemaining--;
            Debug.Log("Wood Remaining: " + WorldGen.woodRemaining);
```

Before optimisation:

```
void Update () {
    Debug.Log("Current Night:" + currentNight);
}
```

After optimisation:

```
using UnityEngine.SceneManagement;

public class VariablesBetweenNights : MonoBehaviour {

    public static int currentNight;

    private void OnEnable()
    {
        SceneManager.sceneLoaded += OutputNight;
    }

    private void Start()
    {
        DontDestroyOnLoad(gameObject);
        currentNight = 1;
        Screen.SetResolution(1920, 1080, FullScreenMode.FullScreenWindow);
    }

    void OutputNight(Scene scene, LoadSceneMode mode)
    {
        if (scene.buildIndex == 1)
        {
            Debug.Log("Current Night:" + currentNight);
        }
    }
```

The second issue I resolved in this pass was more time consuming and required me to re-organise some of the scripts in the code base to allow for me to reset their values to the starting value for that level, meaning that I was able to reset the game without reloading the scene. I started by implementing a new script attached to my GameManager that runs an event when the game would normally reload the scene, see Fig 3. That being when the player falls out of the map or when the player reaches the end of the level. The event then triggers functions in each of the following listed scripts in order to reset the game values:

- MoveRightTemp.cs (Controls player movement)
- RemovePlayer.cs (Hides player when level is completed)
- WorldGen.cs (Controls world generation)
- FollowPlayer.cs (Controls the world collapsing mechanic)
- Score.cs (Controls the game score)

The scripts that are able to trigger a game reset are:

- Buttons.cs (Controls menu buttons)
- KillBoxMovement.cs (Controls the killbox for when the player falls off of the map)

This is the script that I created and attached to my game manager that controls the event and resets a couple GameObjects that do not have a suitable location for the event elsewhere.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class ResetLevelScene : MonoBehaviour
{
    [SerializeField] private GameObject endLevelCanvas;
    [SerializeField] private GameObject upgradeHolder;
    private GameObject wall;
    private GameObject player;

    public static ResetLevelScene m_instance;

    public delegate void Reset();
    public static event Reset OnReset;

    private void OnLevelWasLoaded(int level)
    {
        if(level == 1)
        {
            wall = GameObject.FindGameObjectWithTag("Destruction");
            player = GameObject.FindGameObjectWithTag("Player");
        }
    }

    private void OnEnable()
    {
        m_instance = this;
    }

    public void ResetLevel()
    {
        player.SetActive(true);
        upgradeHolder.SetActive(true);
        wall.transform.position = new Vector3(-15f, -3f, 0f);
        OnReset();
    }

    public void ToggleEndgameScreen()
    {
        endLevelCanvas.SetActive(!endLevelCanvas.activeSelf);
    }
}
```

Figure 3. Event-managed game reset script

1.2 - Conclusion from first pass

After checking the profiler to see if my adjustments made an impact to the project's performance, it was clear that my changes had completely removed the spike in performance where the game was switching between scenes. And my scripts within the "Update.ScriptRunBehaviourUpdate" now average about 1.5% of the total contribution to processing time. This is a big improvement from the spikes of about 20% contribution that I showed in the profiler earlier, see Fig 2. After optimisation, the debugs only run when the value is modified, instead of being run each frame.

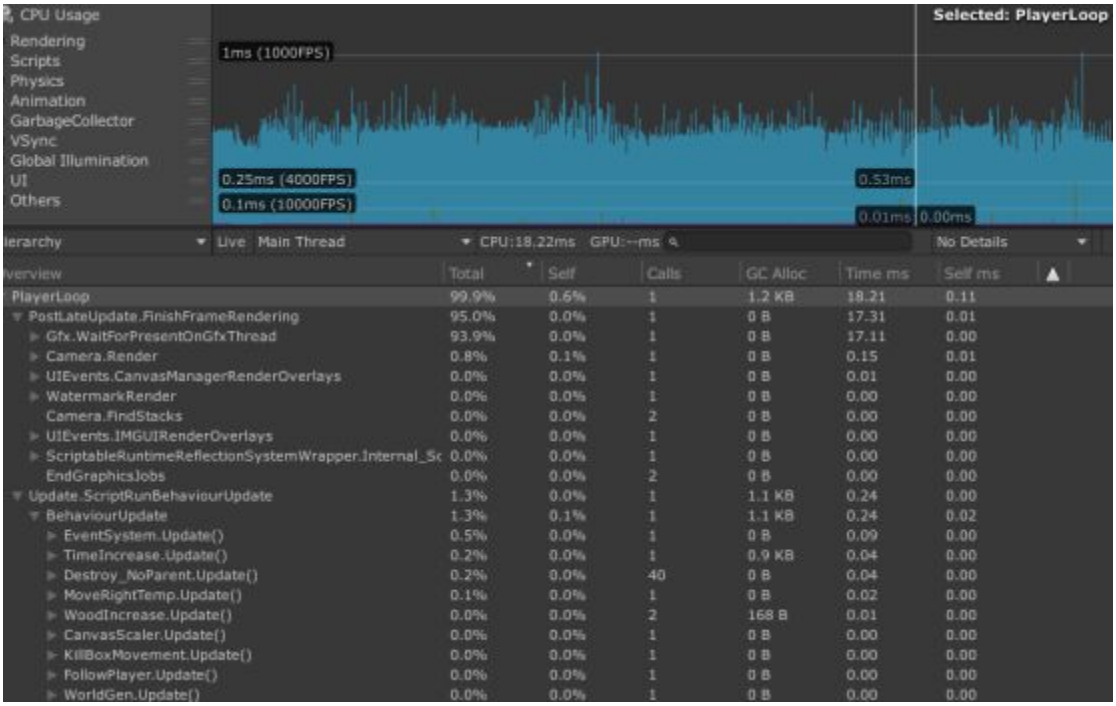


Figure 4. Profiler showing new baseline for the main gameplay loop

2 Identifying issues for a second pass

Now the project is consistent running at below 1ms processing time on the CPU, with occasional spikes of ~1ms. However, within the frame that is highlighted in the profiler screenshot above, see Fig 4, I spotted that my function for destroying world tiles and pickups is being called more than I had anticipated. I realised that due to the nature of an endless runner, I am using a large amount of objects throughout the lifetime of a single level with each of the objects being destroyed with only a short lifespan. If I wanted to expand on this project, more objects being instantiated and destroyed would result in more calls and a lot more garbage cleanup that could create a lot of performance issues and bottleneck.

The way in which I decided to avoid this issue, is to attempt to implement object pooling into the project so that new tiles are generated in bulk at the start of the game being run and objects to be recycled instead of deleted, hopefully resulting in significantly reduced performance hits during runtime. This is also useful for making the project more expandable as the amount of objects that are created at the start of the game would be easily modifiable through the inspector.

2.1 Resolving issues that were outlined

Using my findings from the results of the first pass, I was able to identify an issue with the amount of times I was calling for objects to be instantiated/destroyed. In order to counteract that, I implemented object pooling into the game to replace instantiation, in the middle of the main gameplay loop, and to recycle world tiles to reduce the performance impact and reduce bottlenecks that might appear later on in development.

I achieved this successful implementation by producing a script to manage object pooling and an interface to run a function on each pooled object once it is recycled.

To recycle the objects, I added a script that considers an object 'out of action' if it no longer has a parent, see Fig 6. Therefore, if the object is in the 'ActiveWorldPieces' holder, it will stay active until removed.

I then replaced code that deleted world tiles for code that instead detaches them from the parent object, which resets the object, making it ready for the next time it is called by the object pooler. This modified code is shown below, see Fig 7.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ObjectPooler : MonoBehaviour
{
    [System.Serializable]
    public class Pool
    {
        public string tag;
        public GameObject prefab;
        public int size;
    }

    Singleton

    public List<Pool> pools;
    public Dictionary<string, Queue<GameObject>> poolDictionary;

    void Start()
    {
        poolDictionary = new Dictionary<string, Queue<GameObject>>();

        foreach (Pool pool in pools)
        {
            Queue<GameObject> objPool = new Queue<GameObject>();

            GameObject poolHolder = new GameObject(pool.tag + "Holder");

            for (int i = 0; i < pool.size; i++)
            {
                GameObject obj = Instantiate(pool.prefab, poolHolder.transform);
                obj.GetComponent<ReturnToPool>().parentHolder = poolHolder.transform;
                obj.SetActive(false);
                objPool.Enqueue(obj);
            }

            poolDictionary.Add(pool.tag, objPool);
        }
    }
    public GameObject SpawnFromPool(string tag, Vector3 position, Transform parent)
    {

        if (!poolDictionary.ContainsKey(tag))
        {
            Debug.LogWarning("Pool with tag " + tag + " doesn't exist!");
            return null;
        }

        GameObject objToSpawn = poolDictionary[tag].Dequeue();

        objToSpawn.transform.parent = parent;
        objToSpawn.transform.position = position;
        objToSpawn.SetActive(true);

        IPooledObject pooledObj = objToSpawn.GetComponent<IPooledObject>();

        if (pooledObj != null)
        {
            pooledObj.OnObjectSpawn();
        }

        poolDictionary[tag].Enqueue(objToSpawn);

        return objToSpawn;
    }
}
```

Figure 5. Object pooling script for world tiles

```
public void MakeFall()
{
    Rigidbody2D[] constraintsToRemove = GetComponentsInChildren<Rigidbody2D>();
    for (int i = 0; i < constraintsToRemove.Length; i++)
    {
        constraintsToRemove[i].bodyType = RigidbodyType2D.Dynamic;
        constraintsToRemove[i].constraints = RigidbodyConstraints2D.None;
    }
    StartCoroutine(DisableObject());
}

IEnumerator DisableObject()
{
    yield return new WaitForSeconds(2);
    gameObject.transform.parent = null;
}
```

Figure 7. Disabling objects to trigger 'ReturnToPool'

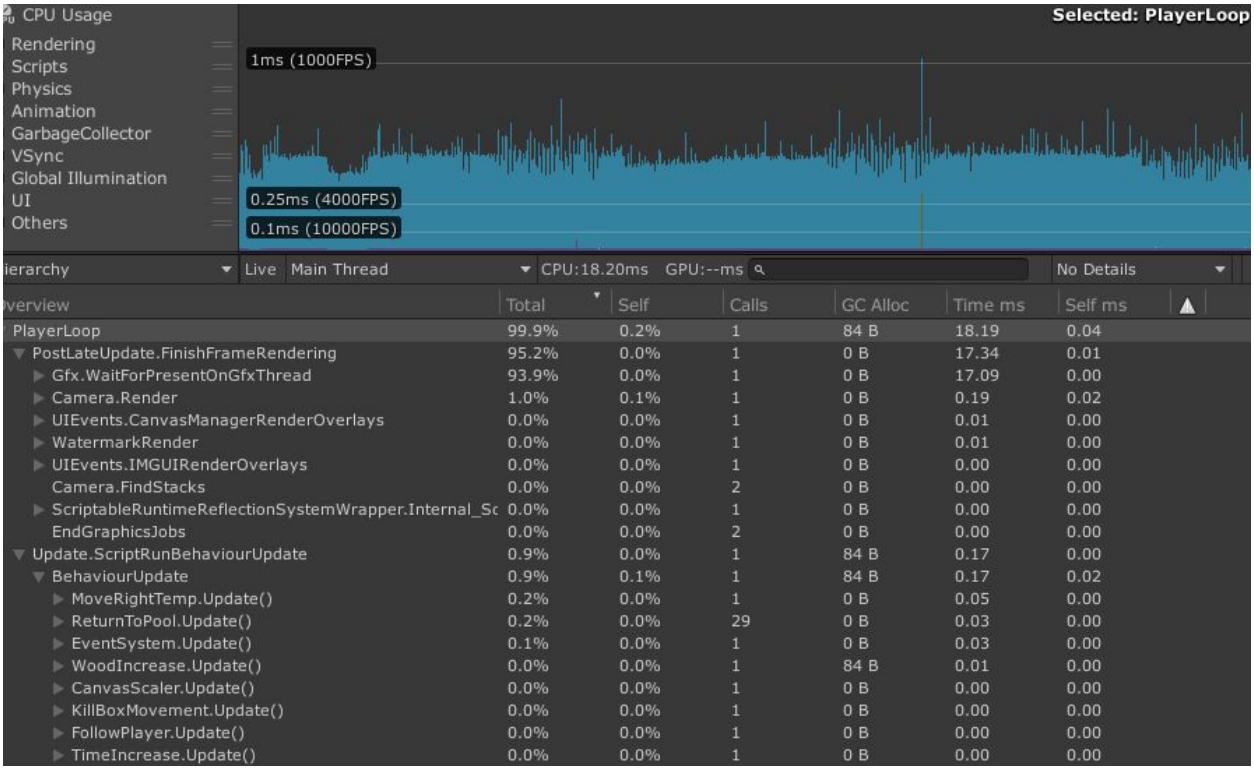
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ReturnToPool : MonoBehaviour
{
    public Transform parentHolder;
    private List<Vector3> startingPositions = new List<Vector3> { };
    private List<Quaternion> startingRotations = new List<Quaternion> { };

    private void Start()
    {
        if (transform.childCount > 0)
        {
            for (int i = 0; i < transform.childCount; i++)
            {
                Transform obj = transform.GetChild(i).transform;
                startingPositions.Add(obj.position);
                startingRotations.Add(obj.rotation);
            }
        }
    }

    public void OnObjectSpawn()
    {
        if (transform.childCount > 0)
        {
            for (int i = 0; i < transform.childCount; i++)
            {
                Transform obj = transform.GetChild(i).transform;
                obj.position = startingPositions[i];
                obj.rotation = startingRotations[i];
            }
        }
    }

    // Update is called once per frame
    void Update()
    {
        if (transform.parent == null)
        {
            if (transform.childCount > 0)
            {
                for (int i = 0; i < transform.childCount; i++)
                {
                    transform.GetChild(i).GetComponent<Rigidbody2D>().bodyType = RigidbodyType2D.Static;
                }
            }
            else
            {
                GetComponent<Rigidbody2D>().bodyType = RigidbodyType2D.Static;
            }
            transform.parent = parentHolder;
            gameObject.SetActive(false);
        }
    }
}
```

Figure 6. Script for returning objects back into their respective holder for organisation

3 Final Conclusion

The change to performance from the second pass is minimal at this point but is surprisingly more than I had expected. The game is currently running at ~0.4ms processing time, see Fig 8, making the game consistently smooth with no noticeable framerate drops or performance issues and no delay/jumps between scenes each time the player dies or finishes a level.

There are still a few visible spikes caused by the garbage collector but these spikes seem to be issues within the project that are either unavoidable or would require a complete overhaul of the project. At the moment, the project sits at about 95% of the processing time for scripts being a result of rendering and not something that I am able to optimise within my codebase.



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PlayerLoop	99.9%	0.2%	1	84 B	18.19	0.04
PostLateUpdate.FinishFrameRendering	95.2%	0.0%	1	0 B	17.34	0.01
Gfx.WaitForPresentOnGfxThread	93.9%	0.0%	1	0 B	17.09	0.00
Camera.Render	1.0%	0.1%	1	0 B	0.19	0.02
UIEvents.CanvasManagerRenderOverlays	0.0%	0.0%	1	0 B	0.01	0.00
WatermarkRender	0.0%	0.0%	1	0 B	0.01	0.00
UIEvents.IMGUIRenderOverlays	0.0%	0.0%	1	0 B	0.00	0.00
Camera.FindStacks	0.0%	0.0%	2	0 B	0.00	0.00
ScriptableRuntimeReflectionSystemWrapper.Internal_Sc	0.0%	0.0%	1	0 B	0.00	0.00
EndGraphicsJobs	0.0%	0.0%	2	0 B	0.00	0.00
Update.ScriptRunBehaviourUpdate	0.9%	0.0%	1	84 B	0.17	0.00
BehaviourUpdate	0.9%	0.1%	1	84 B	0.17	0.02
MoveRightTemp.Update()	0.2%	0.0%	1	0 B	0.05	0.00
ReturnToPool.Update()	0.2%	0.0%	29	0 B	0.03	0.00
EventSystem.Update()	0.1%	0.0%	1	0 B	0.03	0.00
WoodIncrease.Update()	0.0%	0.0%	1	84 B	0.01	0.00
CanvasScaler.Update()	0.0%	0.0%	1	0 B	0.00	0.00
KillBoxMovement.Update()	0.0%	0.0%	1	0 B	0.00	0.00
FollowPlayer.Update()	0.0%	0.0%	1	0 B	0.00	0.00
TimeIncrease.Update()	0.0%	0.0%	1	0 B	0.00	0.00

Figure 8. Final profiler view of the main gameplay loop

Overall, I believe I have made significant optimisations to this project in relation to the baseline I recorded at the beginning of the report. I also believe that I've made this project more expandable and removed some oversights that might have caused issues in the future, had I continued to develop this project.