Optimisation Testing Report

Introduction

This report documents the process of testing the program's performance before and after asynchronous loading techniques have been applied for loading a texture. This is intended as a base identifier for how the game's performance can be improved from the development side.

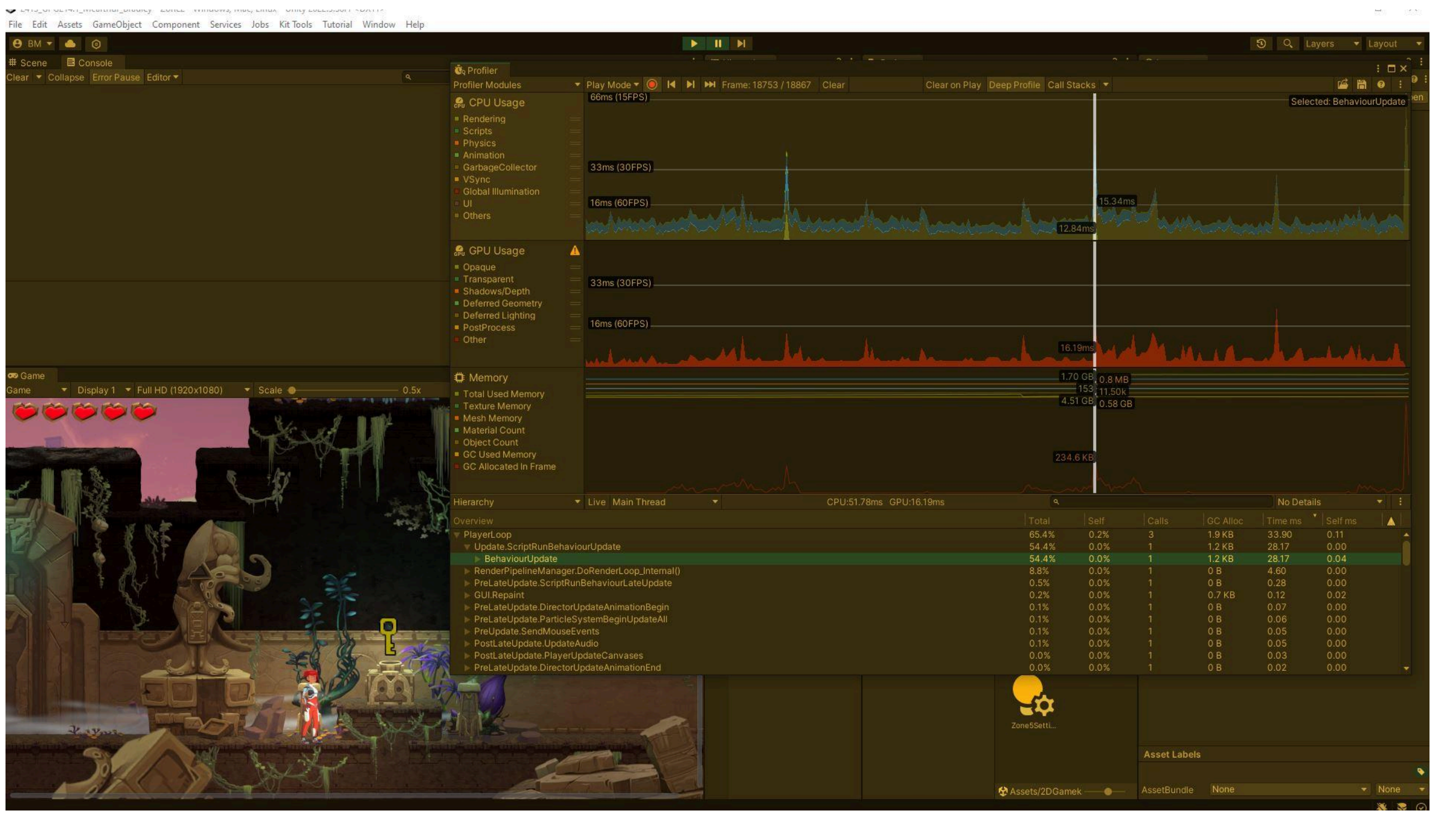
Testing Methodology

The testing was conducted on unity using the built in profiler. I determined the usage of the time percentage using Playerloop and determined the drop or increase based on each method of loading.

Test Cases

The first test was loading a new Key sprite into the scene while the game was running using a key loading script that found and loaded the script from the StreamingAssets folder. The second was using Asynchronous loading to load the texture upon starting and storing it to be used when the key sprite needed to be loaded based on player input.

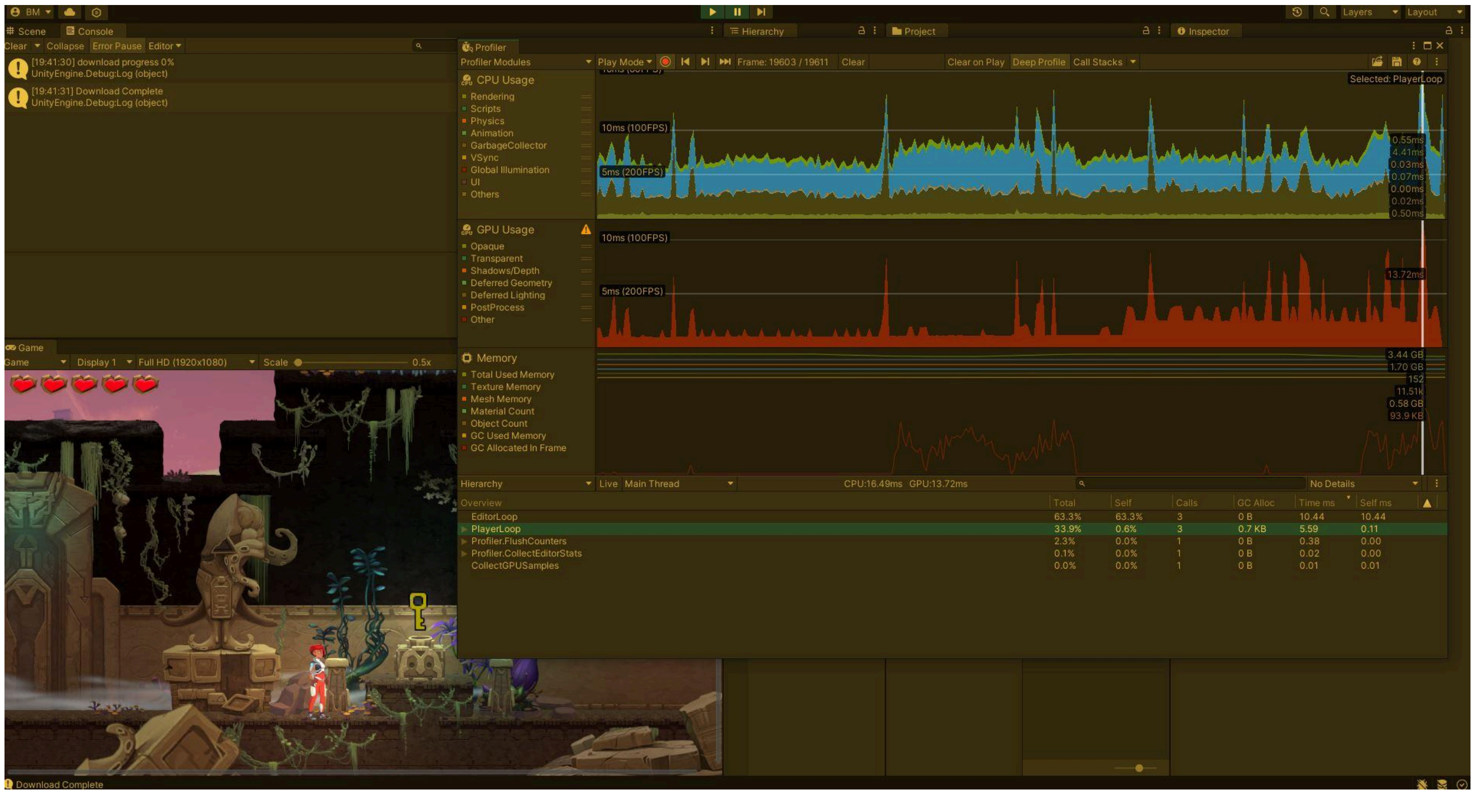
Profiling Results



CPU:51.78ms GPU:16.19ms

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PlayerLoop	65.4%	0.2%	3	1.9 KB	33.90	0.11
Update.ScriptRunBehaviourUpdate	54.4%	0.0%	1	1.2 KB	28.17	0.00
BehaviourUpdate	54.4%	0.0%	1	1.2 KB	28.17	0.04
RenderPipelineManager.DoRenderLoop_Internal()	8.8%	0.0%	1	0 B	4.60	0.00
PreLateUpdate.ScriptRunBehaviourLateUpdate	0.5%	0.0%	1	0 B	0.28	0.00
GUI.Repaint	0.2%	0.0%	1	0.7 KB	0.12	0.02
PreLateUpdate.DirectorUpdateAnimationBegin	0.1%	0.0%	1	0 B	0.07	0.00
PreLateUpdate.ParticleSystemBeginUpdateAll	0.1%	0.0%	1	0 B	0.06	0.00
PreUpdate.SendMouseEvents	0.1%	0.0%	1	0 B	0.05	0.00
PostLateUpdate.UpdateAudio	0.1%	0.0%	1	0 B	0.05	0.00
PostLateUpdate.PlayerUpdateCanvases	0.0%	0.0%	1	0 B	0.03	0.00
PreLateUpdate.DirectorUpdateAnimationEnd	0.0%	0.0%	1	0 B	0.02	0.00

The above data is the initial profiler results from loading the key sprite from its own script. We see a total of 54% usage by the player loop behaviour elements. This isn't only due to key loading but is a massive spike during the frame in which we loaded it.



```
[19:41:30] download progress 0%
UnityEngine.Debug:Log (object)
[19:41:31] Download Complete
UnityEngine.Debug:Log (object)
```

Hierarchy | Live | Main Thread | CPU:16.49ms GPU:13.72ms | No Details

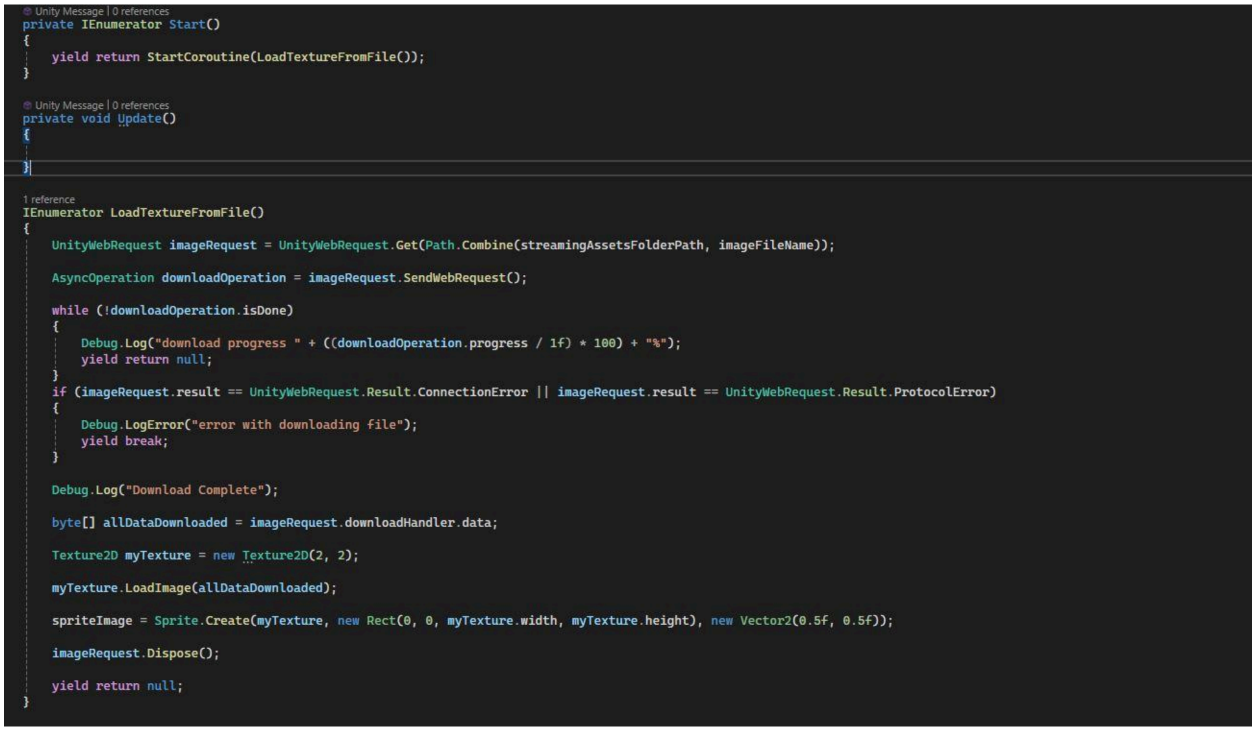
Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
EditorLoop	63.3%	63.3%	3	0 B	10.44	10.44
PlayerLoop	33.9%	0.6%	3	0.7 KB	5.59	0.11
Profiler.FlushCounters	2.3%	0.0%	1	0 B	0.38	0.00
Profiler.CollectEditorStats	0.1%	0.0%	1	0 B	0.02	0.00
CollectGPUSamples	0.0%	0.0%	1	0 B	0.01	0.01

This data is collected after using Async loading to load the Key sprite in which we see a 30% usage on the frames it was loaded. This shows a massive increase in efficiency for this and can be optimised further using other async operations.

Optimisation Strategies

This method of loading Asynchronously allows the data to be more effectively loaded and stored through runtime. It also allows the data to be disposed of when whatever tasks needed are complete.

Below is the Async code for loading the image and converting it into both a texture and sprite stored and ready to be parsed later.



```
Unity Message | 0 references
private IEnumerator Start()
{
    yield return StartCoroutine(LoadTextureFromFile());
}

Unity Message | 0 references
private void Update()
{

}

1 reference
IEnumerator LoadTextureFromFile()
{
    UnityWebRequest imageRequest = UnityWebRequest.Get(Path.Combine(streamingAssetsFolderPath, imageFileName));

    AsyncOperation downloadOperation = imageRequest.SendWebRequest();

    while (!downloadOperation.isDone)
    {
        Debug.Log("download progress " + ((downloadOperation.progress / 1f) * 100) + "%");
        yield return null;
    }
    if (imageRequest.result == UnityWebRequest.Result.ConnectionError || imageRequest.result == UnityWebRequest.Result.ProtocolError)
    {
        Debug.LogError("error with downloading file");
        yield break;
    }

    Debug.Log("Download Complete");

    byte[] allDataDownloaded = imageRequest.downloadHandler.data;

    Texture2D myTexture = new Texture2D(2, 2);

    myTexture.LoadImage(allDataDownloaded);

    spriteImage = Sprite.Create(myTexture, new Rect(0, 0, myTexture.width, myTexture.height), new Vector2(0.5f, 0.5f));

    imageRequest.Dispose();

    yield return null;
}
```

Performance Improvement

As stated above a large amount of runtime performance is optimised through the loading of assets and features Asynchronously. I believe further improvements can be made however using more loading of other components at the same time within the Async loads, for example the audio clips or asset bundles.

Challenges and Lessons Learned

The largest challenge was translating the various amounts of data presented by the profiler, and ensuring its being read correctly. I think it also added a layer of difficulty using a package as oppose to my own game as some elements of what was running were a bit hard to understand throughout the optimisation process (for example all of the animators and character models)

Conclusion

The biggest take away from this process was the efficiency and benefit of Asynchronous loading and its further possible applications in current and future projects. While this small sample showed efficiency there are even more effective implications for optimisation using this method.

Min 300 words.
Includes images of performance testing and optimisations applied.

Word Count:	459