jamk

Optimizing mobile games in a Unity environment

Case: Multiplayer mobile game

Jerry Blåfield

Bachelor's thesis
May 2021
School of technology
Degree Programme in Information and Communications Technology

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Description

Blåfield, Jerry

Optimizing mobile games in a Unity environment, Case: Multiplayer mobile game

Jyväskylä: JAMK University of Applied Sciences, May 2021, 35 pages

Technology, Bachelor's Degree Programme in Information and Communications Technology, Bachelor's thesis

Permission for web publication: Yes

Language of publication: English

Abstract

Mobile games take up majority of the video game market where competition is harder than ever. The performance of the games can determine the size of a game's user base and its marketability. When a game performs badly the video game developers need knowledge of several tools and processes to achieve performance improvements.

The assignor was Zaibatsu Interactive, which is a game development company based in Jyväskylä. Zaibatsu Interactive provided a game project for the research to be implemented on. The emphasis on mobile games specifically was due to the company mostly focusing on mobile game development.

First it was researched how to use the tools in Unity to profile the game on a target device so that optimization targets could be found. Then it was assessed how the most useful optimization targets could be identified in the target game. Lastly some optimization fixes were attempted, and the results were analyzed.

The results showed that even someone without much prior experience in optimization was able to create relatively decent improvements to a game after understanding the game's systems and researching some of the root causes of the problems. This information can be taught to others and used in future development.

Keywords/tags (subjects)

Unity, optimization, mobile game, profiling, rendering, performance, game development

Miscellaneous (Confidential information)

Contents

Figures

1 Introduction

The video game industry keeps growing at an astonishing yearly rate with an estimated 2.69 billion video game players worldwide in 2020 and mobile games make up almost half of the whole market revenue. (Gilbert, n.d.) As video games become a bigger part of everyday life, the competition in the industry has become harder than ever before. Meanwhile the consumers grow more critical of the content they consume. This is especially true for the mobile game industry where players on a wide range of devices play games that often do not require a long attention span. A badly performing game with a long loading time can cause a player to drop the game altogether and in a worst-case scenario for the developer, rate it negatively. The mobile gaming market has become so huge and saturated that consumers who decide to spend their downtime with a mobile game, put a higher value on their quality and performance making it an important part of the game development process. Since many gaming services have moved to online marketplaces and app stores that have user review systems, the users have more power than ever to influence the marketability of a game. If the performance of a video game on the market is not deemed acceptable by the users, the ratings for the game might decrease affecting the purchase and download rates negatively.

This thesis was assigned by Zaibatsu Interactive Oy (Zaibatsu Interactive), a Finnish game development company based in Jyväskylä Finland. The company was founded in 2014, currently has 20 employees and is seeing formidable growth as a company in the past few years. Most games by the company are being developed by outsourcing, subcontracting, co-developing and through partnerships. Even still Zaibatsu Interactive does have some of their own games as well. In addition to game development, Zaibatsu Interactive offers code-for-hire and frontend development services for other companies.

The game project that the research will be implemented on was also provided by Zaibatsu Interactive and is an online multiplayer game for mobile devices. At the time of the thesis the game is under active development by the company. This game was chosen since I had prior experience working on it during a 5-month student internship done for the company.

The objectives of the thesis are to find the best practices for finding performance optimization targets, how to profile games on different devices and what kind of results the optimization will yield.

The thesis is made from a quantitative research standpoint and more specifically focuses on producing a research-based development assignment. To reach the established goals, first a basic understanding of optimization, all the tools and Unity specific information will be researched. After this, the environment for the implementing of the researched information will be set up and tested. When the research is done the profiling and optimization will be implemented on the associated game project to get some authentic experience. Lastly, the results will be summarized, and a conclusion will be discussed.

The researched material is acquired through searches on video game optimization and most of that material is expected to be from technical documents. This material is mostly discovered through search engine and university library database searches. In addition to technical documents some books, articles, tutorials, and conference talks will be used to confirm the information and to get it in a more understandable format. Several sources are used for a single concept and then cross referenced to avoid outdated information and misinterpretation. The sources used were to be as new as possible and when using older information, it was made sure that it was general information that is still relevant at present.

2 Game optimization

To improve the effectiveness of a game's resources and reduce their use overall as much as possible, the process of game optimization is used. While optimizing it is important to avoid affecting the visual quality of the game negatively. The performance of a game dictates the amount of assets and features that can be added to it while staying within an acceptable level of performance. This level is dictated by the market and requires some testing, knowledge, and intuition from the developer if wanting to fully utilize it. As video games are usually meant for entertainment, a game with many features may look good but not be fun to play due to bad performance. (Garney & Preisz, 2011) The need for optimization can vary in many ways depending on the product being developed but the process mostly aims for the same core goals whether wanting to improve performance for the sake of adding more features or trying to make a more smoother gameplay experience. Disregarding the reasons for optimization, a better optimized game will amongst other factors also improve the power usage of the device when playing (Unity, 2017). This in turn is an important factor especially on mobile devices that depend on battery power, and consequently may even affect the length of a user's playing session.

When viewing optimization, it only has a few steps in total and may not seem very complicated. Investigated further the individual problems may quickly get problematic. Due to this, Garney and Preisz (2011) point out that “Optimization should always be done holistically. Look at the big picture first and then drill down until you find the specific problem that is slowing your application” (p. 5). Therefore, it is important to take time in the beginning and not rush at the first problem. First step should be to create a benchmark of the game that can be repeated quickly and consistently so that it is possible to check the baseline and results of the optimizations easily. The next step is to detect the performance problems which are affecting the game the most by using profilers and observations on low end devices. When the target problems have been found, it is measured how and what resources they use, and how to improve them. After the improvements are done, they are measured again and if the results are acceptable, the optimization is done. However, if the performance is still bad, the next problem needs to be found so the optimization process can be started again. (Garney & Preisz, 2011)

It is important to learn how to prioritize, analyze and measure the game properly to avoid unnecessary work that has low returns. A common risk is making too many assumptions of what is causing problems without knowing the details. Optimizing the game too early and only using one device for measuring test results should also be avoided. Using assumptions is important but the problem for beginners is that they might not be completely aware if their assumptions are correct. Both wrong assumptions and optimizing too early can be avoided by benchmarking and profiling the game broadly and accurately. When it comes to the devices used in testing, it is important to use as many of them as possible and always aim for the worst one available. This helps finding the problems that affect the game the most and enables avoiding false results. Users that tend to use higher end devices might not even notice the performance problems which can be game breaking in the lower end. (Garney & Preisz, 2011)

3 Unity

Unity is one of the world’s most popular real-time development platform creating 3D and 2D content of which a majority is games, but it can also be used for creating other interactive experiences. Over 70% of the top 1000 mobile games in 2020 were made using Unity. (Unity Technologies, 2021d)

Unity is free for students and for personal developers whose revenue or funding doesn't exceed $100K during the past year. There are several licenses for professional developers which range in prices, amount of technical support and features. The monthly price for the plus license is $40 while a pro license costs $150 and the Enterprise one is $200. Unity doesn't collect royalties for the games created with it, so if the users have the correct licenses, they will own the full rights to the content they created and will not need to pay any extra costs to Unity. (Unity Technologies, 2021e)

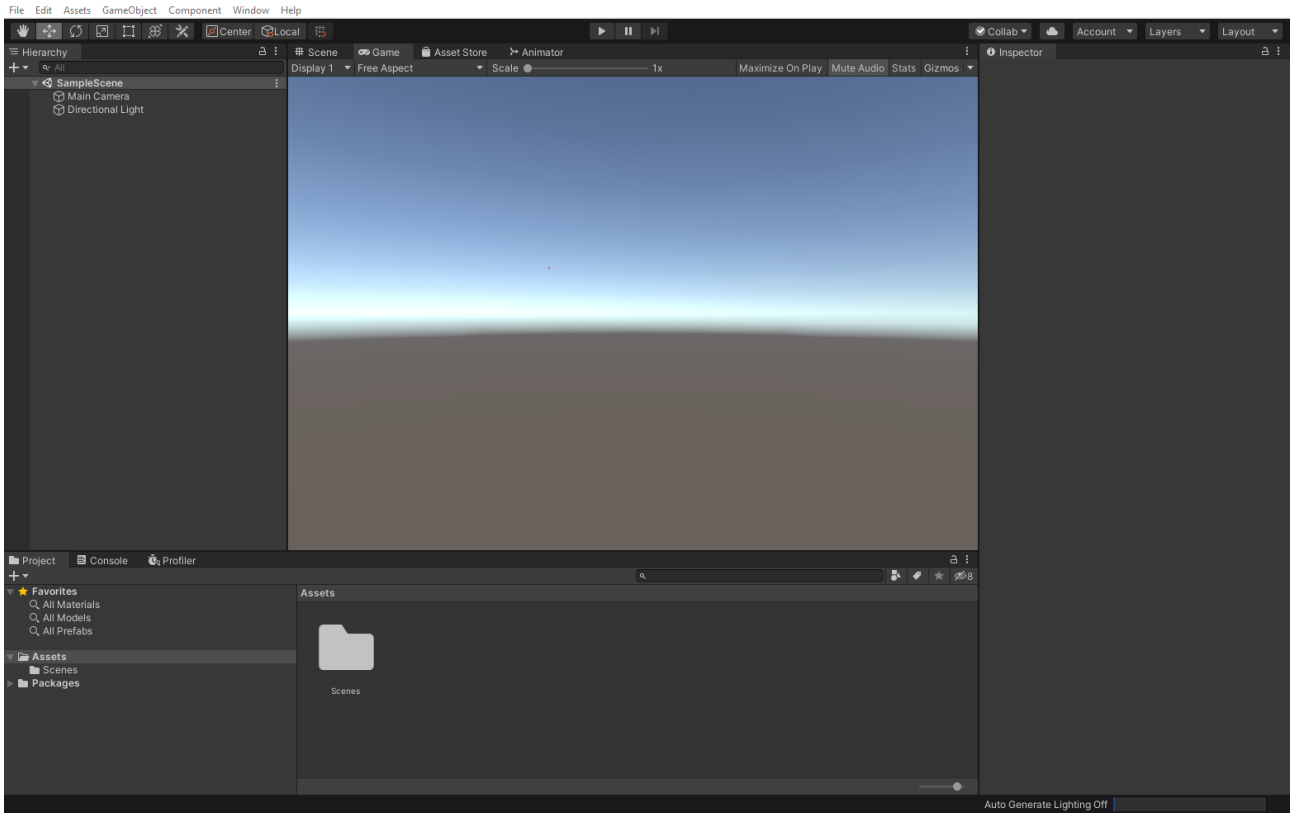
Due to the large amount of users Unity has a very large community built around it with official blogs, forums, Q&A services, live help for professional developers and a thorough documentation (Unity Technologies, 2021a). Unity offers a wide array of free learning content for beginners and professionals, which cover many of the most common features and topics (Unity Technologies, 2021c). The amount of user created learning material is also very ample which makes Unity one of the easiest game engines to start developing with. Unity is the chosen game engine for this thesis since the target game was already being developed with it.

3.1 Unity Editor

Working in with Unity is done mostly in the Unity Editor which has a wide array of features that help in developing a finished game. The main user interface of the editor usually consists of around 5-7 main windows but since it is quite customizable, it differs by user.

Figure 1

An overview of the Unity interface



More windows can be opened and then pulled to the preferred area from the Window dropdown menu in the top toolbar. This is also where the Unity profiler is found, which is not enabled by default and will be explained in more detail in the next chapter. Generally, when developing in Unity the windows which see the most use are the hierarchy, project, console, inspector, scene, and game windows. (Unity Technologies, 2020n)

When opening a scene in Unity a list of all the GameObjects in the scene are listed in the Hierarchy window. From this window the order of the GameObjects can be reorganized and since Unity uses a parenting system the dependencies of the objects on each other can be modified by setting them as parent, child, or sibling objects. All the GameObjects in the hierarchy are children of the main GameObject which is the scene, but others can also be set as parent-child objects below it. All child objects are affected by the parent and sibling objects share the same parent object. Double clicking an object in the hierarchy will bring the scene view to that object and open its details in the inspector window. (Unity Technologies, 2020o) For maximum performance it is commonly

advised to keep the hierarchy structure as simple as possible by avoiding multiple levels of child components if unnecessary (Unity, 2020a).

By default, the middle usually has the Game and Scene views of which the prior shows the rendered version of the game through the main camera in the scene, whereas the Scene view shows an editable view which the developer can freely navigate through (Unity Technologies, 2020n). These views can also be set to be shown side by side to enable doing changes in the scene view that and seeing how they look in the game simultaneously.

The inspector window is where you change the properties of a selected GameObject by adding components like scripts, images, audio, cameras amongst other things. Here selected assets, prefabs and other settings can also be modified. At the top of the inspector, you can rename the selected GameObject, change tags and layers, and even disable from the scene altogether. Being able to flag the object as static helps with improving performance on objects that do not move and is helpful in processes such as lightmapping, which will be explained in a later chapter. (Unity Technologies, 2020p)

All the files included in the project can be found and navigated to from the Project window. This is the main way to get to different assets when needing to modify them, add them to GameObjects or referencing to them. The search bar at the top can help save time by quickly finding specific files in a larger project and can also be used for searching Unity Asset Store assets. Files can also be dragged to and from the Project window to remove or add them to the project. (Unity Technologies, 2020r)

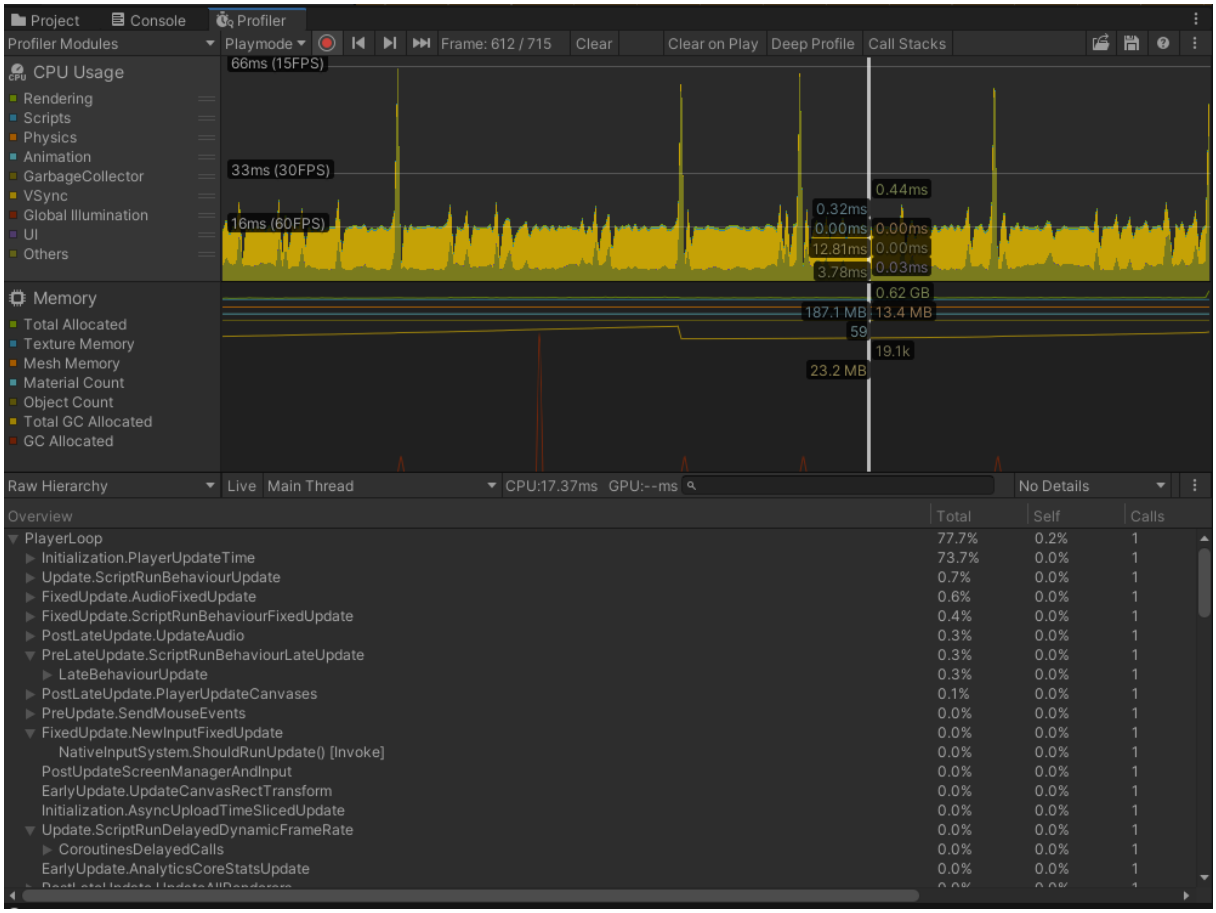
For development and debugging the Console window is one of the most important. This window shows all the messages that are generated by Unity and can also be used for showing debug messages added by developers for additional information on how the game is running and what game is being executed. These messages also get saved to a log file which can be viewed from the Console window menu. The messages that are shown can be cleared, searched through, or filtered in the toolbar at the top of the window. The preferred code editor will be opened to the corresponding point in the associated scripts when either double clicking on a console entry or from the details of a clicked entry at the bottom of the window. (Unity Technologies, 2020b)

3.2 Unity Profiler

For collecting performance data, the Unity editor has an inbuilt profiler component. The profiler is used to record and display performance data and is one of the most important tools in Unity for optimization. It can be used to find specific information on parts of the game that are slowing it down. (Unity Technologies, 2020q)

Figure 2

An overview of the Unity Profiler



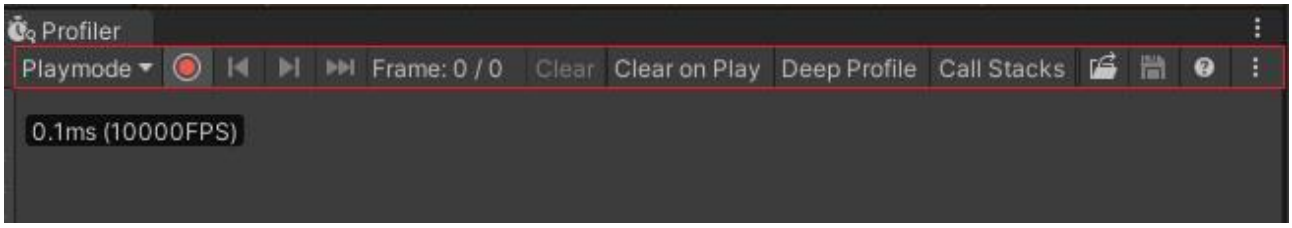
It is possible to choose the number of frames the profiler records from in between 300 and 2000 frames with the default being the former, raising this value however might have an impact on performance. The profiler is used to find out what parts of a game project are slowing it down by seeing the resources and scripts that are being used during the recording. (Unity Technologies, 2020q) This chapter will briefly go through the different parts of the profiler and how it is used for finding optimization targets.

3.2.1 Profiler Controls

The profiler controls are at the top of the profiler window. These are used to navigate through the profiling data and to change some settings which will be explained in further detail below. (Unity Technologies, 2020q)

Figure 3

The controls of the Unity Profiler



From a dropdown on the left side of the controls, the user may choose the device they want to profile on. Next to the dropdown is the record button for starting or stopping recording and arrow buttons for moving frame by frame in between recorded frames. To the left is the frame number indicator which shows the current selected frame. Next are the clear and clear on play buttons which can be used to clear the profiler of all recorded data or set all data to get cleared when the game is launched. The deep profile button enables the recording of all C# script methods for a more detailed view. Call stacks enables the recording of scripting memory allocations. Load is used to open previously recorded profiling data and save is used to save recorded data. Lastly there is the context menu where the settings of the profiler can be modified. (Unity Technologies, 2020q)

3.2.2 Profiler modules

Under the profiler controls are the profiler modules and the charts connected to each one. Profiler modules collect data about different parts of the game being profiled and shows the data in the accompanying charts. By selecting a module, the user can see further info about it in the Module Details panel at the bottom of the Unity Profiler. The modules can be enabled and disabled for getting more specific information and to avoid unnecessary modules affecting the game's performance during the profiling. (Unity Technologies, 2020q)

3.2.3 CPU Usage module

The most important module for getting an overview of the time the game uses on each frame is the one that measures the central processing unit (CPU) usage. This module can be used to pinpoint which other modules should be enabled and unlike the other modules even if it is closed, it will always be active. (Unity Technologies, 2021q) This module's chart shows how much time is spent on every frame on nine different categories which are Rendering, Scripts, Physics, Animation, Garbage Collector, VSync, Global Illumination, UI and Others. These categories can be drag and dropped to a different order and by clicking on the colored legend to the left of the category name, the category's visibility can be toggled. Many of the categories are connected to the other modules which is why this module is so good for finding out which modules should be focused on. (Unity Technologies, 2020c)

Selecting this module will show a breakdown of the time spent on different parts of the game during the currently selected frame. This timing data is by default shown in a timeline format but can also be viewed in a hierarchical table by using a dropdown menu in the top-left corner of the details view. The details pane can also be set to show the newest recorded frame during recording by enabling the Live setting which can be activated from a button next to the module details dropdown menu. (Unity Technologies, 2020c)

3.2.4 GPU Usage module

The GPU usage module shows information about the time spent on graphics processing in the game and will be disabled by default due to its high overhead. This module can only be used in the Playmode or builds of the game. According to the Unity documentation GPU profiling is supported for Windows, macOS, Linux, PlayStation 4, Xbox One, Switch, WebGL, Android, iOS, tvOS and Tizen platforms. The charts for the GPU Usage module are divided into seven categories, which are Opaque, Transparent, Shadows/Depth, Deferred PrePass, Deferred Lighting, PostProcess and Other. The categories give data about time spent on each pipeline's rendering. When selecting the GPU Usage module, the details pane will show time data for the selected data in a hierarchical table. Unlike the CPU Usage module this module cannot show the timing data in a timeline format. The data had three columns, Total which shows the time spent on a particular function in percentages, DrawCalls which shows how many times a function was called during the frame and GPU ms

which shows the total amount of time spent on the function in milliseconds. (Unity Technologies, 2020f)

3.2.5 Rendering module

For measuring detailed rendering information and statistics about the rendered objects in the scene the profiler has a rendering module. The charts in this module display four different values. The first data displayed is the number of batches done per frame. The SetPass Calls data shows the number of times a shader pass is used in rendering GameObjects. Unity shaders can contain these shader passes to allow it to render GameObjects differently on every pass. The last two graphs are used for finding how many triangles and vertices the GameObjects in the scene include. (Unity Technologies, 2020l) This data can be used to establish if a scene has rendering problems due to containing too many or unoptimized objects (Unity Technologies 2020i). The data from the charts is also displayed in the details pane alongside more detailed rendering data such as the resolution of the screen, texture counts, VRAM memory usage, shadow data and computation information. From the “Open Frame Debugger” button a running game can be frozen on a single frame and in the frame debugger the individual draw calls of in the scene can be viewed in greater detail. (Unity Technologies, 2020l)

3.2.6 Memory module

An important module for profiling load times and memory usage is the Memory module, which can be used for finding out information like the number of loaded objects, how much memory they use in each category and the number Garbage Collector allocations per frame. This module has six chart categories. The Total Allocated category shows how much memory the game has used in total. Texture Memory and Mesh Memory categories show how much memory has been used for textures and meshes. Material Count and Object Count categories tell the number of material and object instances in the game. If the Object Count value always keeps rising the game has a problem of not deleting old GameObjects while creating new ones. Lastly the Total GC Allocated category shows the total amount of memory used by the Garbage Collector heap while the GC Allocated category shows it’s per frame memory allocation. It is important to remember that the values shown in the Memory Profiler are not the same as shown in the device’s task manager due to

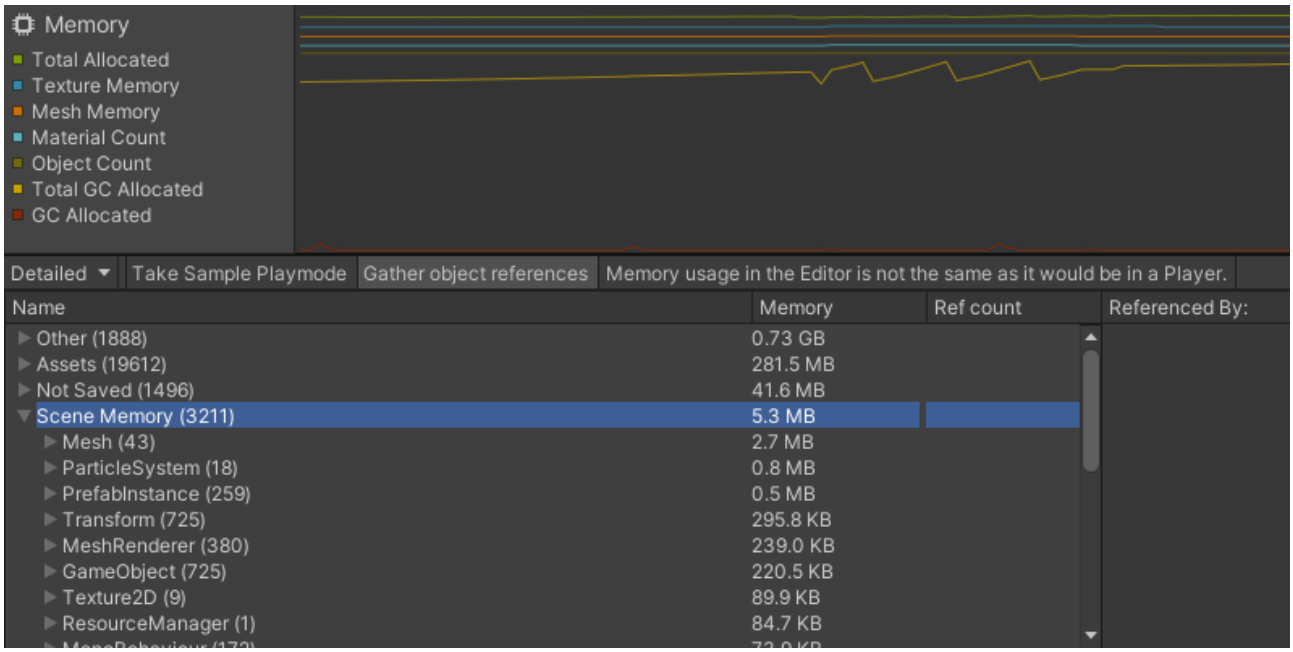
the profiler only tracking the memory usage connected to the game and not the whole system. (Unity Technologies, 2020g)

When selecting the Memory Module there are two views available in the details pane, the simple and detailed view. The simple view shows an overview of the memory usage in the game for the current frame. Firstly, at the top of the view is the total memory used which is an accumulated value of all the other memory data that has been collected. The next data shows how much memory is allocated in native Unity code. Mono a garbage collected total heap size and used heap size managed code uses. GfxDriver shows and estimated amount of memory that is being used on rendering, textures, shaders and meshes. The audio and video values show and estimated memory usage value for the audio and video systems. Lastly the profiler value shows how much memory the profiler itself is using. (Unity Technologies, 2020g)

The detailed view can be used to collect a sample of detailed memory usage data for a specific target by clicking the Take Sample button. Taking a detailed sample takes a long time and cannot be used to get real time memory usage data. The collected sample data is shown in a tree view which shows how memory is used in a very detailed way. By enabling the Gather Object references setting the detailed view will also show what objects are referencing the memory. (Unity Technologies, 2020g)

Figure 4

Memory snapshot in the detailed view of the memory profiler



The tree view is divided in to five categories of objects that use memory. The Other category is made up of objects that are not assets, GameObjects or components. The Not Saved category had all the objects that have been marked to not be saved to the current Scene and can only be explicitly destroyed by the owner of the object. The “Builtin Resources” include Unity Editor resources like Shaders which have been added to the always included list of the graphics settings in Unity. The Assets category has all the assets which are reference in user or native code. The last category is Scene Memory which comprises of the objects and attached component that are in the current scene. An object can be directly highlighted in the Scene view by clicking on it in the Assets and Scene Memory categories, which is useful for fixing memory problems in specific objects. (Unity Technologies, 2020g)

3.2.7 Other modules

The audio module shows the amount of audio sources being played in the scene, how many audio channels are being used and the CPU and memory usage of the audio (Unity Technologies, 2020a). The video module works in a much similar way showing the total number of video sources in the scene, how many are currently playing and their resource usages (Unity Technologies, 2020s). The Global Illumination module measures the time spent on the lighting in a scene and other parts of

the Realtime Global Illumination system in Unity (Unity Technologies, 2020e). The user interface (UI) modules can be used to find out how time and resources are used on coordinating and rendering the user interface (Unity Technologies, 2020m). There are also Physics modules for measuring the number of physics components in the scene like collisions and joints (Unity Technologies, 2020k). Lastly there are Network Modules the use of which has been deprecated. (Unity Technologies, 2020q) These modules are used for finding very specific problems in parts of the game which are not important in the constraints of this thesis.

4 Unity specific optimization

When discussing optimization in Unity it is important to mention some well some commonly known issues that developers often make to know what to keep an eye on during the optimization process. It should be acknowledged however, that every optimization should be looked at case specifically and what might work on one game may reduce the performance in a different one. Thankfully due to the large popularity and userbase of Unity a very wide array of tutorials and tips exist in this topic and there is a very high chance that the problems found have been discovered and fixed by someone already. This chapter will go through some of these tips while focusing mostly on the ones that are the most important to mobile games.

4.1 General suggestions

Developing mobile games can have a difficult time balancing between the shorter development schedules and the need for optimization. In such an environment it is more important than ever to find out if the optimization is necessary in the first place. (Unity, 2016) As mentioned in previous chapters, testing and profiling should be done often and with proper target devices. This leads to a better assessment of budget needs for the optimization and has the potential for saving large amounts of time (Garney & Preisz, 2011).

Using the proper tools can speed up the process of profiling and help in finding the root of the optimization problems. The Unity profiler is limited in this sense especially in its capabilities to find more detailed memory problems and getting important information from outside the game such as how long it takes for the game to launch on a device. (Unity, 2016) For these more specific profiling purposes there exists a multitude of tools to use. For iOS devices the Instruments tools found

in XCode can be utilized (Apple Inc., 2021). In Android studio the Android Profiler can be used to measure the app performance with more detail (Android for Developers, 2020). For memory there is a Memory Profiler tool which can be added to a Unity project with the built-in Package Manager. This tool can be used to take snapshots of development builds for a much more detailed way of profiling memory usage. (Unity Technologies, 2019b)

4.2 Optimizing scripts

Working on a new project while using good coding practices can help to avoid later optimization needs. However, if the need for optimizing the code in a project later arises it is vital to first profile what causes the biggest impacts on performance. (Unity Technologies, 2020h) One commonly missed easy improvement in scripts is removing empty “Start()” and “Awake()” functions which although having small performance impacts by themselves, can in large game projects stack up to becoming a noticeable issue (Unity, 2020a). Similarly, if there is code that is only necessary for debugging during the development phase it is ideal to remove or comment them out of the finished product especially if the functions are called every frame. Code functions that are only needed during development can also be set to only working in the Unity Editor or for example a development build by enclosing them in an if-statement the contents of which will be omitted when the game is built. Customized if-statements for this purpose can be utilized by configuring them in the Player settings in Unity. (Unity, 2020a; Unity Technologies, 2020j)

Most games have several features where the game needs to instantiate and remove game objects frequently and a great way to reduce the load on the CPU and memory allocation issues to use object pooling. (Unity, 2020a; Unity Technologies, 2020h) What object pooling means is the preloading of game objects into a pool of a predetermined size and then hiding them when they are not in use. This way it is possible to reuse the objects while saving the CPU from constantly having to make new objects and destroying the old ones. At the same time the memory usage will be at a constant level and the objects do not need to be rendered when not in use. (Unity Technologies, 2021b) A good example of using object pooling would be weapons that shoot a similar projectile multiple times in quick succession. If the number of allowed projectiles on screen at once can be limited, they can be preloaded into an object pool.

4.3 Rendering optimization

Often when graphics performance has been poor and it is then optimized without sacrificing quality, it uses resources somewhere else. If the GPU load is slowing the game down but there is capacity in the CPU or memory to lighten that load, it is a good idea to use practices that make the resource usage more balanced. This way the performance of certain components will get slightly worse, but the overall performance may be noticeably improved. Balanced resource usage will also reduce the possibility of a singular component being susceptible to thermal throttling which will also reduce power usage. Naturally graphics performance can also be improved by reducing the overall resource usage altogether. One way to get large improvements for both the CPU and GPU is to reduce the amount of lighting that must be calculated by baking lightmaps. Using this technique, a scene's non-movable objects can have their static lighting be precalculated thus reducing the performance impact significantly. Especially on low end devices rendering dynamic lights for every frame can be very expensive so it should be avoided and reduced as much as possible. When lightmapping it is also important to remember that obviously it does not work with moving lights. (Unity Technologies, 2020h; Unity Technologies, 2020i)

In a Unity game a draw call is issued every time a GameObject needs to be drawn on screen and it tends to create CPU overhead due to its resource-intensiveness. Using draw call batching is a great way to optimize games where too many resources are used unnecessarily for drawing similar graphics. To avoid this, GameObjects that use the same materials can be batched together to speed up their rendering. For moving objects, a technique called dynamic batching is done to transform all similar GameObject vertices into the scene using the CPU. By enabling dynamic batching in the Player settings, Unity could achieve this automatically if the materials used in the game are shared between GameObjects. As such it is recommended to use them in as many different GameObjects as possible. For stationary GameObjects that have been marked as static a technique called static batching can be used. Whereas dynamic batching used the CPU for calculating vertices faster its static correspondent uses memory to store combined geometry. This can have serious memory usage impacts if used on many objects. As such it should be reviewed beforehand if the cost on memory can be allowed when comparing to the rendering speed benefits. (Unity Technologies, 2020d)

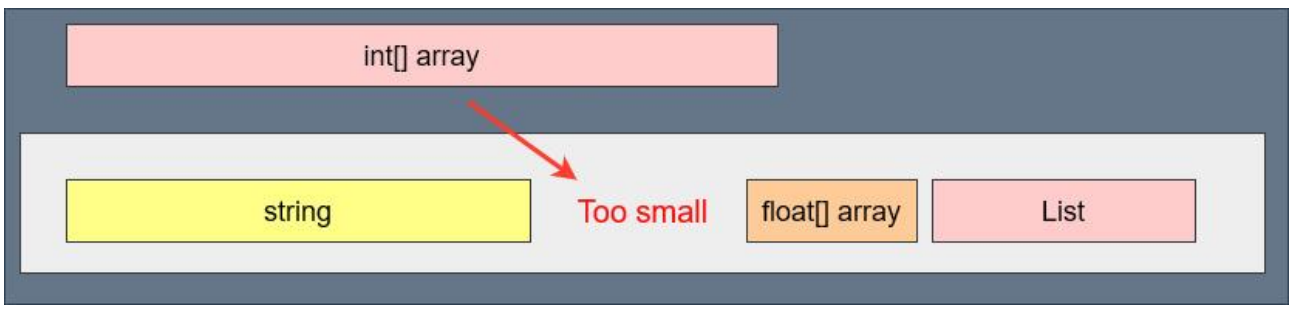
Optimizing textures, materials and other such assets is also very important both for the graphical performance and the build size of a 3D game. Using compressed textures and reducing their resolution as much as possible can free up the memory footprint of the game and increase rendering speeds significantly, especially when a scene contains many surfaces. It is also good to enable generating mipmaps in the texture import settings so the GPU can lower the resolution even further for smaller triangles. This should not however be done for UI elements and 2D graphics which are meant to be rendered in their original sizes. (Unity Technologies, 2020i)

4.4 Memory usage

The way that Unity handles memory and garbage collection can cause severe impacts on performance. The managed heap to which memory is allocated by the memory manager can often expand much more easily than shrink. This is due to every object created in managed code being allocated to the managed heap in which the garbage collection can cause memory fragmentation that locks the memory from being used after being freed. The fragmentation occurs due to freed memory being the same size and in the same location as the previously allocated memory, which can still be surrounded by other memory objects. Due to this a "gap" of memory is created which cannot be used by an object that is bigger than the freed space. This can cause the heap to expand to accommodate the large memory allocation. The reason this happens is due to the managed memory being like a chain of blocks where an uninterrupted block of memory is required for an allocation to succeed. (Unity Technologies, 2018)

Figure 5

Heap memory fragmentation visualized



Note. The gap in between the memory allocations which has been created by garbage collection, is too small for the new array that is being allocated. Due to which the memory manager has no choice but to expand the total size of the heap.

When a memory heap has been expanded, Unity does not usually shrink it back down even if mostly empty. This is done to avoid the need for repeated expansion in case of other large memory allocations. Unity does eventually return the empty memory back to use by the operating system but the period during which it happens can differ depending on the system. However, the managed heap does not return the address space back to the operating system which in a 32-bit program can cause the programs memory to run out, which results in the program crashing. (Unity Technologies, 2018)

One of the simplest ways of cleaning up, reducing load times, and releasing runtime memory is reducing your game's build size. Many of the optimizations and procedures in this chapter will help with this since the quantity of resources and code in the game are proportional to runtime memory usage. (Unity Technologies, 2019a)

To reduce the memory impact of assets it is important to acknowledge certain tips. Since Unity does not release the memory in use by GameObjects if disabled or set to null, it is important to destroy objects when possible. To avoid short term objects from being allocated on the memory heap and later garbage collected they can be set as structs instead of classes which should be used with long-term objects. The short-term objects should also be object pooled when applicable. (Unity Technologies, 2019a) When coding Enumerators it is important to keep in mind that they do not clean up their memory until they exit. Thus, avoiding endless Coroutines is to be avoided so that the memory can be released and reused as soon as possible. (Unity Technologies, 2018) To further reduce the impact of assets the unused channels in meshes should be removed. Unused keyframes should be removed as well. By enabling the maxLOD setting from the quality settings can remove higher detail meshes from the build. Accessing the material property of renderers should be avoided since it creates a clone of the material even if nothing is assigned to it. (Unity Technologies, 2018)

5 Implementing the optimization

5.1 The target game project

The game project used in this thesis was provided by Zaibatsu Interactive and is an online multiplayer game, developed for Android and iOS devices. As explained in the introduction the game is

developed with Unity and more specifically in the version 2019.4.16f1. The game has both single- and multiplayer game modes but playing requires an internet connection.

The game has two servers and due to always requiring an internet connection it was chosen to restrict the profiling to the test server. This was done since the live server would not have been directly modifiable, working there would have resulted in outdated results and might have affected the players who were playing the game. Working on the test server was also not ideal and due to time constraints, it was not possible to create a separate environment where to test the game without it requiring a connection and being affected by the active development. This was a risk for getting accurate results and slowed down profiling and testing considerably.

5.2 Optimization process

Before the optimization itself started, the developers who worked on the project were asked what parts of the game seemed to perform the worst and several pointers were discovered. Everything discussed was however taken with a grain of salt since when beginning optimization, it is vital to profile thoroughly before doing anything else (Unity Technologies, 2020h). Even still this information was used during the profiling to either verify or invalidate if they were good targets to focus on. The discussion also brought up the topic of memory usage, since external quality assurance results had observed that devices with low RAM memory had crashed in specific parts of the game. These results meant that the game's memory usage had to be improved upon or the on-release game would have to be restricted to exclude the lowest tiers of devices.

5.2.1 Profiling

In the general profiling phase, the Unity profiler was used with the game. A few profiling runs on the Unity editor revealed some problems with CPU and memory usage in several places, but to get accurate results an Android build of the game had to be made and run on a target device. The game was built through Unity's build and run system, then being installed, and run on a Huawei Honor 7 phone through a USB cable connection. The device was running Android 6.0, had a Hisilicon Kirin 935 processor and 3,0 GB of RAM memory. The device was not exactly low tier, but also nowhere near a high tier level, and was still decided to be sufficient for the results of this thesis.

Running builds through USB from Unity was tricky at best. The device had to be set to the “photo transfer” USB mode for the build to be opened by Unity. When the build was working a sudden disconnect could result in the profiler no longer recognizing the running build even though the “autoconnect profiler” setting was enabled, and thus a new build had to be made for the profiler to function again. Although making a new build and running it only took a few minutes, the number of builds that needed to be done was quite large due to the setbacks. When making a good working build succeeded, it was ideal to do as much profiling as possible and save the data for analyzing later. The saved profiling data does not directly convey which part of the game the saved frames were recorded from, so it is important to keep a consistent way of naming the files. For these files it was decided to use a naming convention which informed the observer of which game scene it was recorded in, the event during which the spikes happened, any differences in settings and when it was recorded. For this project only one person was observing the data so the naming did not have to be perfect but in the case of working with multiple people a better naming convention should be agreed upon before profiling. Even when working alone, having to go back to record data that has already been recorded due to not knowing where it was stored, would be a big loss of both time and resources.

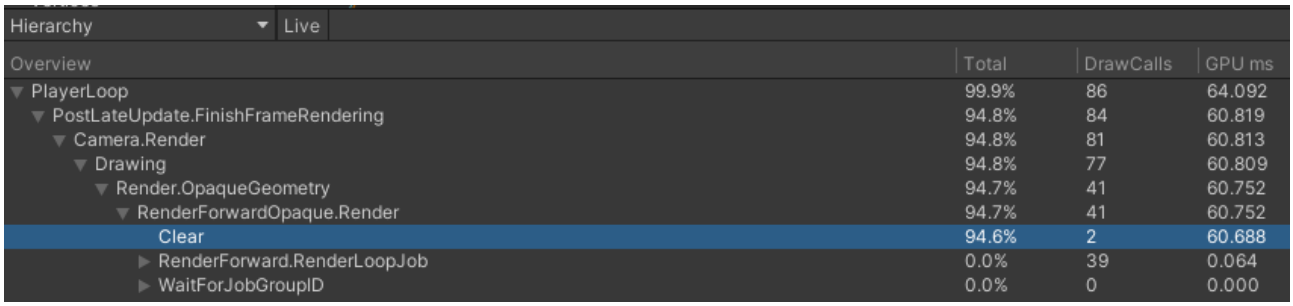
When it comes to the results of the profiling, the game was mostly working within acceptable levels but there were still many problems in many parts of the game. The biggest spikes in CPU and memory usages were understandably in places where the game scene was changed, or new resources were loaded into the scene. Several spikes in both CPU and GPU usage were found in seemingly normal parts of the core gameplay. The GPU usage was highest in areas where many effects were being used. High memory usage and a large build size were also associated with old assets that were still in the game even though they were always disabled due to either having become irrelevant after removing features or being left over from features which got reworked. This problem got fixed by removing and packing older assets and improving the currently used ones, resulting in a much lower memory usage in the game overall. It is worth mentioning this fix due to its effects on other performance improvements, even though it was a fix made separately from this thesis project.

One of the first problems that was easily visible, were spikes after using an item in the game that creates a lot of particle effects. Looking at the CPU usage hierarchy pointed to a function called

FinishFrameRendering, which took nearly all the time during a single frame. This function meant that the CPU was caught waiting for the GPU to finish rendering, resulting in slowing the game to a halt for a corresponding amount of time. Since this meant the problem was GPU bound, the next step was to look at the GPU usage during the same frame.

Figure 6

Rendering issue diplayed in the profiler hierarchy

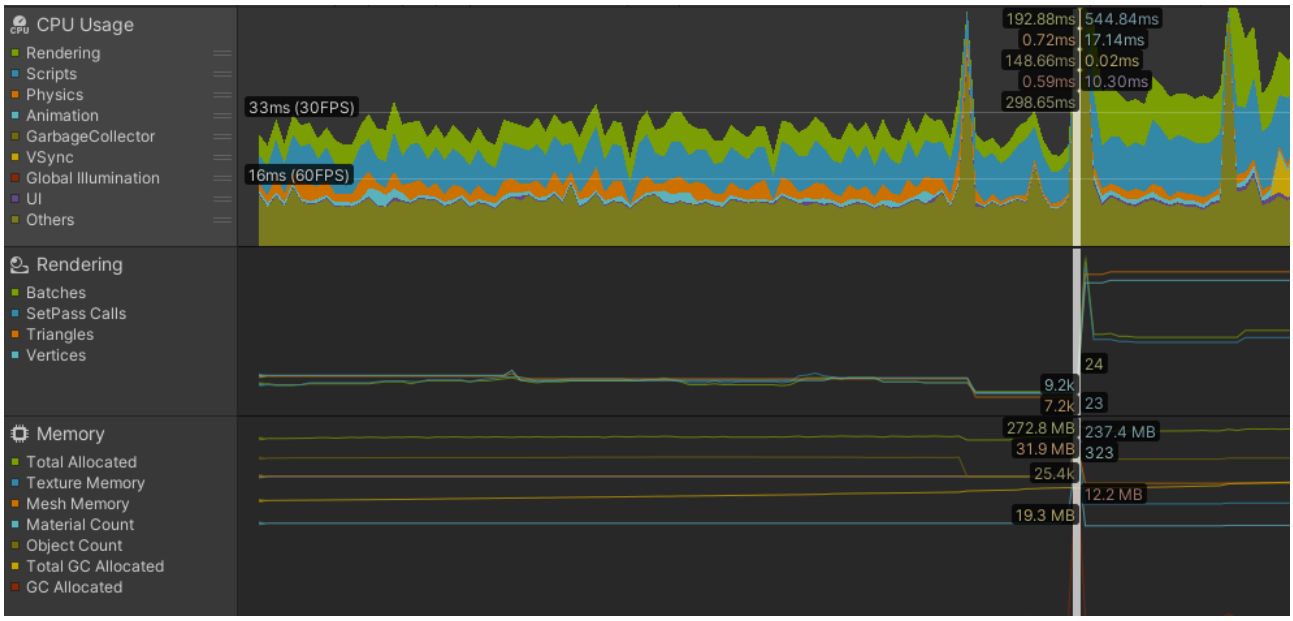


The hierarchy view of the GPU usage module showed a PostLateUpdate.FinishFrameRendering using 94.8% of the total time during that frame. Since this is the main function for the rendering pipeline it meant that the rendering would have to be optimized by reducing draw calls and rendered objects and choosing the most well performing rendering settings. Similar issues were found when UI popups were opened on top of the previous menu increasing the amount of draw calls by around 25% resulting in a 40% drop in framerate on the used device.

In a scene that showed the scoring after a match in the game, two scenes were always loaded on top of each other while disabling most GameObjects from the previous scene and then using the remaining geometry as a background. This resulted in a higher memory usage and a lot of rendering work that could potentially be avoided. While loading this scene the game stuttered for more than a second on one frame due to the badly performing scripts.

Figure 7

Spike in the profiler when loading two simultaneous scenes



Looking at the rendering during this spike, the number of vertices jumps from about 15 thousand to 54 thousand within the span of two frames. This is massive increase and one of the highest number of vertices in any scene in the game. During these frames the CPU is also allocating unused memory back into the memory pool, so the CPU is using a lot of resources both for rendering and garbage collecting. There are many ways in which the performance could be improved here but the difficult part is finding out how to reduce it with as little work as possible. Fixing the rendering issues might require a bigger operation where the scenes being used could be limited to a smaller portion of the original scenery to greatly reduce the number of rendered surfaces. Another way to improve the scene could be to put results view into a prefab and move to a view where only it alone is rendered. Doing this might however be very time consuming and could take an amount of time that is not worth the results. When it comes to memory usage and garbage collection, they could be improved by spreading the operations over multiple frames. Alternatively, the code that resulted in the garbage collection could also be moved to where both scenes are closed. By doing this the most noticeable performance impacts would be in a loading period where a slight stutter might already be expected. The issues with the scenes were chosen to be further analyzed and will be explained more in the next chapter.

5.2.2 Further analysis

Examining the issue chosen in the previous chapter further, the hierarchy view of the CPU usage profiler revealed that a ResultsController.Start function was responsible for most of the resource usage in the badly performing frames. In this function a multitude of actions were performed. Firstly, a system that allowed players to socially interact with each other was set to connect to the characters. After this the camera and the objects required for the scene were moved into predetermined positions that were referenced from root GameObjects in the background scene. Another function was also called to load the results of the match, which while executing called many other functions and primarily seemed to be the most demanding part of the script. Also, right before the new scene was loaded the moving objects in the previous scene were disabled and left untouched otherwise. Due to this the process' impact on the CPU was quite minor, but all the objects were still using memory. As such, the disabled GameObjects still had some impact on memory usage and even though impacting less when comparing to the whole game level in the background, it was still using enough memory for optimization to be worthwhile. The process for modifying the code that disabled the objects into one that destroys them seemed straightforward enough, so it was kept as an optimization choice for later.

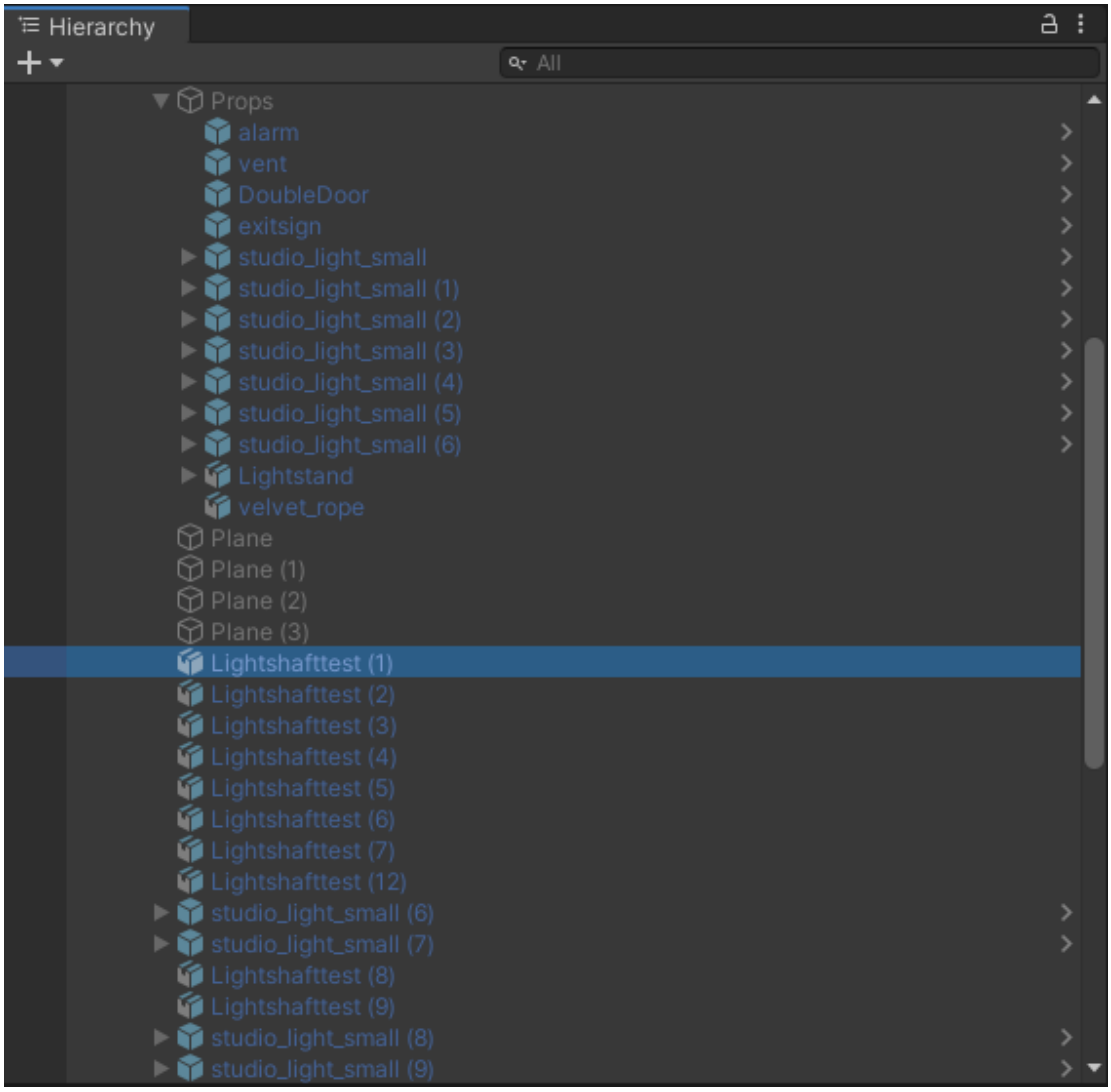
For spreading out the amount of garbage collection done, it was considered to try out incremental garbage collection, which is a feature in Unity that spreads garbage collection over multiple frames to avoid occasional spikes in CPU usage. Generally, it is a good practice to write code that avoids allocating memory in a way where these spikes are created but when time is a concern the incremental garbage collector can offer a viable solution. (Echterhoff, J. 2021) If this is done however, it is important to profile the changes it has on other parts of the game as well to make sure nothing else breaks down as a repercussion.

Upon further inspection both loaded scenes that created issues, were already lightmapped and the static terrain in the game level was already batched together into one entirety. Due to this, other venues for improving the rendering performance had to be investigated. Viewing the GameObjects in the scene hierarchy revealed large number of disabled old versions of objects, test particles and other such objects that were used in the development in the past. It became apparent that these objects might have some impact on memory if they ended up in the final game builds. To resolve their possible effect on the game they could be set to not be included in the

build in by using if-checks in the scripts. However, a much cleaner and easier way would be to move them to a folder outside of the game resources included to clean up the scenes while avoiding losing any assets that may still have a use in the future.

Figure 8

Disabled GameObjects left over from development



Note. The large number of unused objects could have serious impacts on the game if accidentally left in the game during a non-development build.

The scene used in the background also had a lot of resources that were not destroyed and some of them were even not disabled, even though not having any significant purpose due to the camera being locked into a spot where they could not be seen. As a result, thousands of vertices being rendered outside the viewable area. It was suspected that this was one of the biggest explanations

for the abnormally number of rendered surfaces, large CPU and GPU usage during the scenes in question.

5.2.3 Improving performance

Summarizing the ways found to potentially improve the performance during the problem scene were as follows:

- trying out incremental garbage collection
- removing old development GameObjects from the scene
- destroying GameObjects that are not needed after the second scene is loaded
- disabling objects from outside the camera view
- cleaning up expensive code

As the incremental garbage collection required only changing one of the project settings, it was the simplest process for attempting to optimize the game. After the setting was enabled, a new build was made and tested. Comparing the profiling results from the baseline to after the change, showed a very small improvement to the worst spikes in CPU usage while simultaneously several small spikes appeared due to the garbage collection happening during a wider number of frames. Since the overall performance during the spike did not improve and the benefits from spreading out the garbage collection didn't seem very valuable, the thought of using this method was dismissed and the changes to the settings were reverted.

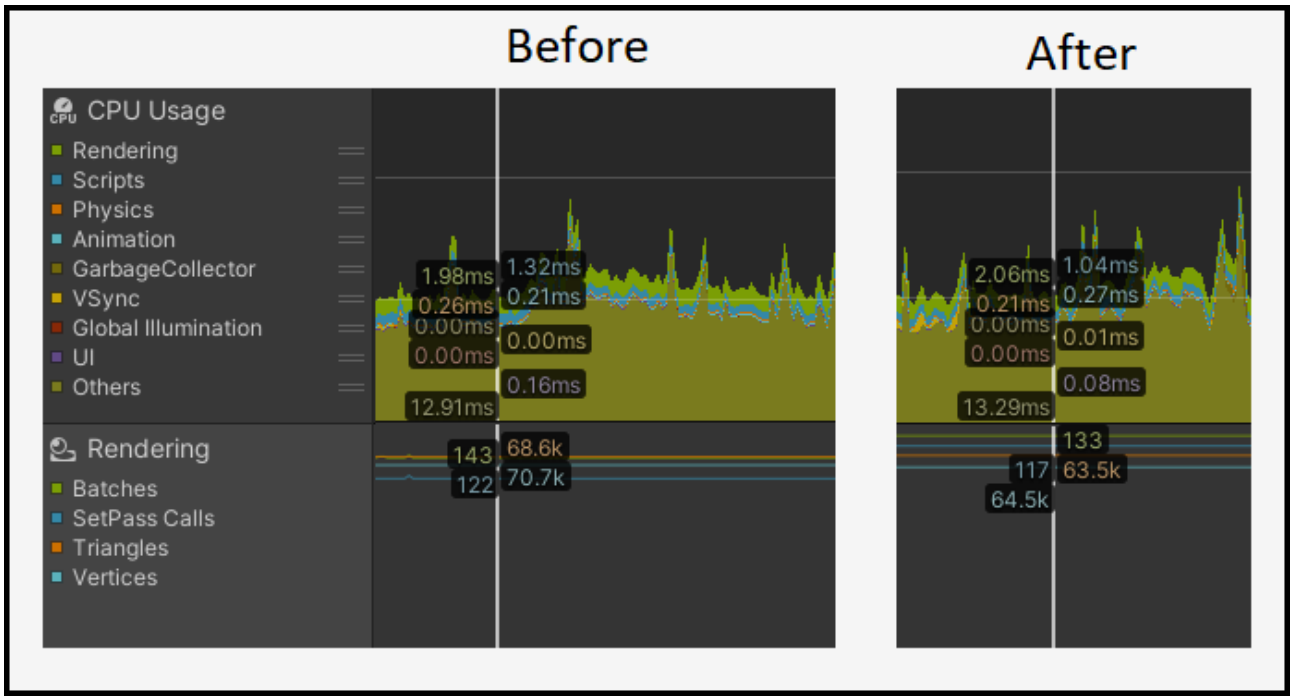
Testing removing the old development GameObjects from the scene was quite simple as well, since for testing purposes they could be deleted and then if no performance increases were achieved, acquired back to where they were by returning discarding the changes using version control. After deleting the objects, the game was built and profiled the same way as the baseline and the memory usage was compared. The removal of the objects resulted in a memory usage improvement of up to 30 megabytes and naturally also reduced the total number of objects in the scene. This was considered as a reasonably quick and easy improvement, which also did not have many risks of breaking other systems. As such, it was a viable candidate to possibly be implemented into the final game in the future. After the improvement results were recorded and confirmed the changes were saved to a separate branch and the original Unity project was reverted to an unmodified version to prepare the game for the next optimization test.

Destroying the GameObjects that were still using memory due to only being disabled required several code changes. Firstly, testing only changing the code where the objects were disabled to destroy them instead, predictably resulted in several errors since functions from background operations still tried referencing the destroyed GameObjects. Due to this several lines that checked if the objects had been destroyed were added to the connected functions, which enabled the code to run without any obvious problems. The results of the changes were quite minimal, and the memory usage decrease seemed much smaller than expected. Most likely this was due to the low number of objects that were being disabled in the code. In other words, the scene still had a lot of unnecessary enabled objects that had to be destroyed or disabled for any noticeable performance improvements. During this testing it became evident that the packing of assets made by other developers had lowered memory usage in a newer branch so much that after the game was updated to that new version, this fix became redundant. Hence, fixing memory usage in the scene by modifying code was not pursued further. Overall, this was also a good opportunity for noticing the importance of good communication with the other developers and staying aware of what was going on with the simultaneously ongoing active development of the game.

Reducing the CPU usage and improving rendering performance by disabling GameObjects that were outside of the camera when viewing the results was initially very simple to test by doing it in the Unity editor. Although this kind of testing didn’t produce results comparable to the original baseline profiling data, the editor allowed immediate feedback on whether the disabled objects would improve performance. By running the game in the editor and disabling the GameObjects manually it was observed that the number of rendered surfaces was understandably reduced by a large amount.

Figure 9.

Data from before and after optimization



Note. The number of vertices and triangles that are rendered in the baseline is thousands more than in the data from after the fix.

It became apparent that to implement the fix to the game more permanently would require to further identify exactly which components where unnecessary during the second scene. The objects could also be bundled together and then disabling them as a unit in the code together with the other objects that were already being disabled in the preexisting code. Due to lacking both time and expertise this was decided not to be implemented during the thesis, but instead the problem and potential fix were documented for the purpose of creating a task for a future implementation.

5.2.4 Optimization results

The results of the optimization showed several improvements to memory usage, rendering performance and a reduced CPU usage. Although individual optimizations resulted in quite minor improvements, put together they could produce a much more noticeable result. Cleaning up unused assets from the scenes not only improved the performance of the chosen issue it also influenced the gameplay happening before and after it. However, even with all the achieved improvements

they would still have required more optimization work, which most likely would have included a larger operation in changing the whole way the scenes were built.

The most notable results were made with improving rendering performance. The number of rendered objects was quick and easy to reduce due to many neglected old features that had not been cleaned from the final release build yet. Due to this it was easily the best improvement that succeeded.

The most difficult and time-consuming optimization was improving memory since it required going through large amounts of code and testing the changes was slow. This was due to other changes fixing the problem that already existed preventing any tangible results. With the small changes that made it into testing phases, a large enough improvement was not observed as to justify the large amount of time spent on them.

6 Conclusion

The objectives of the thesis were to learn how to recognize optimization targets, how to profile on target devices and how to implement the optimizations into a mobile game developed in Unity. The research and work done, although very helpful were slowed down quite a lot by not having much prior experience in many of the topics. Due to this a much too large portion of the thesis process was used trying to identify new concepts that were vital for a deeper understanding of the issues analyzed.

In the beginning of the implementation process not having prior experience caused missing the intuition and experience needed for quick and effective optimization actions and directly affected the results that were produced. Innocently jumping into fixing optimization problems before assessing if the fix was too complicated to repair in the allotted timeframe, resulted in the same blunders that the research material was warning learners not to make. Learning from these mistakes can thankfully help in gaining an understanding of how to convey the learned procedures to other novices in the future. So, although the errors might have restricted the contents of the thesis by some amount, it can still be used to produce noticeable benefits both for myself and the assignor company. Spending a lot of time on research also helped a lot by teaching the basics about

the supposedly complicated concepts. As a result, the profiling and optimization that was done covered a much larger number of different techniques than what was originally considered.

Almost all the different parts of optimization could have been studied at the length of a separate thesis or more, so choosing what to include as essential was extremely difficult. A lot of material was left out and would require a much longer time to research and learn.

When gathering information about the optimization tools and processes, the Unity documentation was used extensively due to it easily being the most up to date source of information. When using official documentation, the risk of wrong interpretation by a middleman was eliminated. To help find real life examples and easier explanations some supporting sources were used. Most YouTube videos that were used as sources were made as official Unity material and allowed a more visual representation of how optimization was done. To avoid risks of outdated advice, the contents of secondary sources were compared with the up-to-date documentation, which always pointed out if a feature had changed significantly from the previous versions.

When it comes to the actual optimizations that were done, since they were not implemented all the way into the final game during this process the optimization results were more in the form quick tests and planned optimization tasks. Still, most of the tests made their way into working test builds which can certainly display the results of fully implementing them quite efficiently. When continuing the optimization process the performance that was freed could even be used in optimizing other tasks if needed.

References

Android for Developers. (2020, October 12). *Measure app performance with Android Profiler*. Retrieved May 7, 2021 https://developer.android.com/studio/profile/android-profiler

Apple Inc. (2021). *Xcode Features*. Retrieved May 7, 2021. https://developer.apple.com/xcode/features/

Garney, B., & Preisz, E. (2011). *Video Game Optimization*. Boston, MA: Course Technology.

Gilbert, N. (n.d.). *Number of Gamers Worldwide 2021/2022: Demographics, Statistics, and Predictions*. FinancesOnline. Retrieved March 18, 2021. https://financesonline.com/number-of-gamers-worldwide/

Echterhoff, J. (2018). *Incremental Garbage Collector.* Unity. Retrieved May 7, 2021. https://resources.unity.com/developer-tips/incremental-garbage-collector

Unity. (2016, June 28). *Unite Europe 2016 – Optimizing Mobile Applications* [Video]. YouTube. https://youtu.be/j4YAY36xjwE

Unity. (2017, October 27). *Unite Austin 2017 – Writing High Performance C# Scripts* [Video]. YouTube. https://youtu.be/tGmnZdY5Y-E

Unity. (2020a, July 6). *Optimization tips for maximum performance – Part 1 |Unite Now 2020* [Video]. YouTube. https://youtu.be/ZRDHEqy2uPI

Unity. (2020b, July 6). *Optimization tips for maximum performance – Part 2 |Unite Now 2020* [Video]. YouTube. https://youtu.be/EK8sX8oCQbw

Unity Technologies. (2018, March 5). *Understanding the managed heap*. Retrieved May 10, 2021. https://docs.unity3d.com/2019.4/Documentation/Manual/BestPracticeUnderstandingPerformanceInUnity4-1.html

Unity Technologies. (2019a, March 6). *Memory Management in Unity*. Retrieved May 7, 2021. https://learn.unity.com/tutorial/memory-management-in-unity#5c7f8528edbc2a002053b599

Unity Technologies. (2019b, December 3). *Memory Profiler*. Retrieved May 7, 2021. https://docs.unity3d.com/Packages/com.unity.memoryprofiler@0.2/manual/index.html

Unity Technologies. (2020a). *Audio Profiler module.* Retrieved April 28, 2021. https://docs.unity3d.com/2019.4/Documentation/Manual/ProfilerAudio.html

Unity Technologies. (2020b). *Console window*. Retrieved May 4, 2021. https://docs.unity3d.com/2019.4/Documentation/Manual/Console.html

Unity Technologies. (2020c). *CPU Usage Profiler module*. Retrieved March 24, 2021. https://docs.unity3d.com/2019.4/Documentation/Manual/ProfilerCPU.html

Unity Technologies. (2020d). *Draw call batching*. Retrieved May 7, 2021. https://docs.unity3d.com/2019.4/Documentation/Manual/DrawCallBatching.html

Unity Technologies. (2020e). *Global Illumination Profiler module*. Retrieved April 28, 2021. https://docs.unity3d.com/2019.4/Documentation/Manual/ProfilerGI.html

Unity Technologies. (2020f). *GPU Usage Profiler module*. Retrieved March 24, 2021. https://docs.unity3d.com/2019.4/Documentation/Manual/ProfilerGPU.html

Unity Technologies. (2020g). *Memory Profiler module*. Retrieved March 24, 2021. https://docs.unity3d.com/2019.4/Documentation/Manual/ProfilerMemory.html

Unity Technologies. (2020h, October 12). *Optimizing for Performance*. Retrieved May 4, 2021. https://learn.unity.com/project/optimizing-for-performance-2019-3?uv=2019.3

Unity Technologies. (2020i). *Optimizing graphics performance.* Retrieved May 7, 2021. https://docs.unity3d.com/2019.4/Documentation/Manual/OptimizingGraphicsPerformance.html

Unity Technologies. (2020j). *Optimizing Scripts*. Retrieved May 4, 2021. https://docs.unity3d.com/2019.4/Documentation/Manual/MobileOptimizationPracticalScript-ingOptimizations.html

Unity Technologies. (2020k). *Physics Profiler module*. Retrieved April 28, 2021. https://docs.unity3d.com/2019.4/Documentation/Manual/ProfilerPhysics.html

Unity Technologies. (2020l). *Rendering Profiler module*. Retrieved April 24, 2021. https://docs.unity3d.com/2019.4/Documentation/Manual/ProfilerRendering.html

Unity Technologies. (2020m). *UI and UI Details Profiler*. Retrieved April 28, 2021. https://docs.unity3d.com/2019.4/Documentation/Manual/ProfilerUI.html

Unity Technologies. (2020n). *Unity's interface*. Retrieved May 4, 2021. https://docs.unity3d.com/2019.4/Documentation/Manual/UsingTheEditor.html

Unity Technologies. (2020o). *The Hierarchy window*. Retrieved May 4, 2021. https://docs.unity3d.com/2019.4/Documentation/Manual/Hierarchy.html

Unity Technologies. (2020p). *The Inspector window*. Retrieved May 4, 2021. https://docs.unity3d.com/2019.4/Documentation/Manual/UsingTheInspector.html

Unity Technologies. (2020q). *The Profiler window*. Retrieved March 18, 2021. https://docs.unity3d.com/2019.4/Documentation/Manual/ProfilerWindow.html

Unity Technologies. (2020r). *The Project window*. Retrieved May 4, 2021. https://docs.unity3d.com/2019.4/Documentation/Manual/ProjectView.html

Unity Technologies. (2020s). *Video Profiler module*. Retrieved April 28, 2021. https://docs.unity3d.com/2019.4/Documentation/Manual/profiler-video-profiler-module.html

Unity Technologies. (2021a). *Community*. Retrieved March 18, 2021. https://unity.com/commu-nity

Unity Technologies (2021b). *Introduction to Object Pooling*. Retrieved March 27, 2021. https://learn.unity.com/tutorial/introduction-to-object-pooling#

Unity Technologies. (2021c). *Learn Unity*. Retrieved March 18, 2021. https://learn.unity.com/

Unity Technologies. (2021d). *Our Company.* Retrieved March 18, 2021. https://unity.com/our-company

Unity Technologies. (2021e). *Plans and pricing.* Retrieved March 18, 2021. https://store.unity.com/#plans-business