

Aleksi Lehtola

Optimizing Unity Projects



Bachelor of Business Administration

Business Information
Technology

Spring 2018



KAJAANIN
AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Tiivistelmä

Tekijä(t): Lehtola Aleksi

Työn nimi: Unity Projektien Optimointi

Tutkintonimike: Tradenomi, tietojenkäsittely

Asiasanat: optimointi, Unity, pelimoottori, ohjelmointi

Opinnäytetyön tavoitteena oli perehtyä Unity-pelimoottoria käyttävien projektien optimointiin. Työn toimeksiantaja toimi Virtual Frontiers Oy, jonka projektia työn aikana optimoitiin Mac-laitteille toimivaksi.

Tämän opinnäytetyön teoreettinen osa käy läpi optimoinnin periaatteita, Unity-pelimoottorin optimointityökalujen käyttöä ja tälle moottorille ominaisia ongelmakohtia ja ratkaisuja. Siinä myös tutustutaan yleisiin ongelmakohtiin pelien suorituskyvyssä ja haetaan niihin ratkaisuja.

Optimointiprosessin osalta tärkeimmät periaatteet ovat profilointi, optimoinnin priorisointi ja suorituskyvyn seuranta. Profilointiin käytettävät työkalut ovat Unity Profiler, Frame Debugger ja Memory Profiler. Unity Profilerilla seurataan yleistä suorituskykyä, Frame Debuggerilla voi perehtyä tarkemmin GPU:n toimintaan ruudun piirtämisessä ja Memory Profiler näyttää tarkasti, miten pelin käyttämä muisti jaetaan eri objektien välille. Yleisimmät suorituskyvyn ongelmat ovat ruudunpäivitys piikit, joka ruudun prosessit, roskankeräyksen piikit, muistin käyttö ja latausajat.

Työn toiminnallisessa osuudessa käydään läpi projektin työstämisessä ilmenneitä ongelmia ja kerrotaan erilaisista ratkaisuista, joihin päädyttiin. Projektin työstämisessä tuli vastaan seuraavat ongelmat, UI:n suorituskyky, tekstuurien muistinkäyttö ja piirtokutsujen suuri määrä. UI:n suorituskyky parannettiin jakamalla se usealle kankaalle, tekstuurien kokoa pienennettiin ja piirtokutsuja vähennettiin lisäämällä objekteille laatutasoja ja käyttämällä occlusion culling -tekniikkaa.

Opinnäytetyön tuloksena syntyi dokumentti, joka toimii tietolähteenä optimointiprosessia aloittavalle henkilölle. Työosuuden tuloksena yrityksen projekti saatiin alustavasti toimimaan Mac-laitteilla ja sen jatkokehittäminen jatkuu kirjoittajan toimesta.

Abstract

Author(s): Lehtola Aleksii

Title of the Publication: Optimizing Unity Projects

Degree Title: Bachelor of Business Administration, Business Information Technology

Keywords: optimization, Unity, game engine, programming

The objective of the thesis was to study the optimization of Unity game engine projects. The client of the thesis is Virtual Frontiers Ltd, a Finnish game company, whose game project was optimized for Mac devices.

The theoretical part of this thesis deals with the principles of optimization, the optimization tools provided by Unity game engine, the issues found on this platform and solutions for them. It also goes into detail on the common performance issues found in video games and seeks solutions for them.

The main principles in the optimization process are profiling, prioritizing optimization and measuring the performance. The tools used for profiling are the Unity Profiler, the Frame Debugger and the Memory Profiler. The Unity Profiler is used to measure the overall performance, the Frame Debugger is used to debug the rendering process and the Memory Profiler shows how the Memory Usage of the project is spread among the different objects, The most common performance issues are framerate spikes, every-frame costs, garbage collection spikes, memory usage and loading times.

The operational part of the thesis focuses on optimizing the performance of a project and on the unique issues found during the work and on the solutions reached for those issues. The following problems were faced while working on the project; The performance of the UI, the memory usage of textures and the high number of drawcalls. The performance of the UI was improved by spreading it on multiple canvases, the size of the textures was made smaller and the number of drawcalls was brought down by adding levels of detail to models and by using occlusion culling.

As a result for the thesis, a document was created that works as a source of information for those starting on an optimization process. The assignment project reached acceptable performance on Mac devices and the work on it continues.

Table of contents

1	Introduction	1
2	Optimization principles	2
2.1	Profiling	2
2.2	Prioritizing optimization	2
2.3	Measure the effects of changes	3
2.4	The 5 classes of optimization	3
2.4.1	Spikes	3
2.4.2	Every-frame costs	4
2.4.3	Garbage collection spikes	4
2.4.4	Loading time	5
2.4.5	Memory usage	5
3	Profiling Unity	7
3.1	Connecting the Profiler	7
3.1.1	Making a build for profiling	7
3.1.2	Profiling external devices	8
3.1.3	Profiling mobile devices	9
3.2	The Unity Profiler	9
3.2.1	Using the Profiler	10
3.2.2	Reading the Profiler	11
3.3	The Frame Debugger	12
3.3.1	Using the Frame Debugger	12
3.3.2	Reading the Frame Debugger	13
3.4	The Memory Profiler	14
3.4.1	Using the Memory Profiler	14
3.4.2	Reading the Memory Profiler	15
3.5	Benchmarking	16
4	Optimizing Unity	18
4.1	Unity UI	18
4.2	Memory management	18
4.2.1	Garbage collector	19
4.2.2	Manual garbage collection	19
4.2.3	Object pooling	20
4.2.4	Pre-allocating memory	20
4.3	Reducing drawcalls	21

4.3.1	Static game objects and batching	21
4.3.2	Frustum culling.....	21
4.3.3	Occlusion culling	22
4.3.4	LOD levels on 3D objects	24
4.4	Optimizing scripts	25
4.4.1	Preventing spikes	25
4.4.2	Caching	25
5	Optimizing the Virtual Frontiers project	26
5.1	Benchmarking.....	26
5.2	Occlusion culling.....	27
5.3	The UI	27
5.4	Terrain issues	27
5.5	Terrain streaming.....	29
6	Conclusion	30

List of Symbols

LOD – Level of Detail. Refers to the amount of details on a 3D object.

Drawcall – A single task performed on the GPU. A single frame consists of multiple draw-calls.

RAM – Random access memory is the memory on which a game is loaded while running.

VRAM – Video random access memory is on a graphics card and is preserved for graphical effects.

1 Introduction

A massive framerate drop in the middle of an action-packed moment or having to wait for ages in loading screens can throw the player out of the gameplay experience. These are common issues in videogames and are often the result of poor optimization (Turner, Schell 2017).

Optimization refers to the process of making your game run better. Usually the main reason for optimization is to make the gameplay smoother (Turner, Schell 2017), or to make the game more available to a wider audience by having it run better on lower end devices.

When it comes to optimization, every project is unique and has different challenges which need different solutions (San Filippo 2015). Profiling the project should always be the starting point. Find out where the issues lie and start the optimization process from there (Echterhoff, Riber 2013). However, there are common and well-known issues in the Unity game-engine. Existing guidelines can also be found from which the optimization process is easier to begin.

The first part of this thesis focuses on some of these guidelines and provides information on profiling Unity projects. It should help you find a place to start the optimization process from.

The second part of the thesis tells about the optimization work I did during my internship in Virtual Frontiers. I was given the task to work on the performance of their project while porting it to Mac devices.

2 Optimization principles

No matter how grand or small the optimization process is intended to be, there are some general principles and good practices that should be followed. This chapter aims to introduce some of these practices and to provide information on more common areas of optimization. Many issues this chapter introduces have their own dedicated chapter later in the thesis.

2.1 Profiling

To know what is the issue with the performance of a project, profiling is needed. Profiling assists in understanding the unique issues found in a project, and should always be the starting point for optimizing a project (San Filippo 2015).

The first step is to figure out whether your project is GPU or CPU bound (San Filippo 2015). This means which of the two components needs to wait for the other one to finish performing its tasks before moving on to the next frame. Figuring out by which component a project is bound helps to find the optimization tasks with the most performance gain. Profiling has its own dedicated chapter later that goes through the act of profiling and the tools used for it.

2.2 Prioritizing optimization

After the optimization tasks have been determined and before the process of optimization is started, it is advisable to take some time and consider which area of optimization is the priority (San Filippo 2015). This is done by comparing the approximated time an optimization job would take to the expected performance gain of said job.

Through this method a prioritized list can be made. This list then works as a guideline for the entire process of optimization. This also helps to tackle the low hanging fruits first and to gain the most performance with the least effort (San Filippo 2015).

2.3 Measure the effects of changes

Whenever a task of optimization is completed, the project should be profiled and the data stored. The stored data can then be compared with a previous data allowing for closer inspection of the state of the whole optimization process. This then helps to rearrange the priority list, should it be necessary, and assists in keeping the whole process organized (San Filippo 2015).

To accurately measure the changes between optimization tasks, benchmarking is often used. Benchmarking has its own dedicated chapter later in the thesis.

2.4 The 5 classes of optimization

Aaron San Filippo, programmer and co-owner of the game studio Flippfly, divides optimization issues into four main categories in his presentation on Race the Sun Optimization at Unite 2015. These categories are spikes, every-frame costs, garbage collection spikes and loading time.

Memory size has been added as an additional class. This is a lesser issue on modern PCs and powerful gaming machines, but is still relevant on weaker Mac machines and mobile devices (Echterhoff, Riber 2013).

This chapter goes through the classes and provides information on what causes them. Possible solutions for these issues can be found in a later chapter.

2.4.1 Spikes

Spike is a sudden drop in the framerate of a game. This is noticed when the screen suddenly stops and doesn't move for a noticeable time frame. This can break the immersion of the player or cause him to make a mistake he wouldn't have otherwise made (San Filippo 2015). Spikes can be seen in the profiler as a high point in otherwise stable framerate (Picture 1).

Spikes are mainly caused by complex calculations or difficult operations performed during a single frame (Turner, Schell 2017). Spikes are an issue in high-intensity games which

need a steady framerate and high control over the game to feel good, such as driving or shooting games.



Picture 1. Framerate spike in the profiler

2.4.2 Every-frame costs

Every-frame costs are the calculations and operations that are run every single frame (San Filippo 2015). These can be, for example, physics calculations, running AI behavior or handling animations of characters.

Every-frame costs slow down the general framerate of the game. They are the little things that slow the game down and make it feel less fluid (San Filippo 2015). If a game just generally runs poorly, this is the area that needs work.

2.4.3 Garbage collection spikes

Garbage collection spikes are framerate drops specifically caused by the garbage collection system of Unity (San Filippo 2015). These can be huge framerate spikes and are easily noticed while playing the game. The spikes happen when the memory garbage limit is met and the collection runs, cleaning up the unnecessary objects from the memory (Turner, Schell 2017). Their frequency is mandated by how much garbage the game generates each frame. The frequency can be slowed down by generating less garbage during runtime.

The only way of preventing these spikes completely is to generate no garbage during runtime (Kjems, Pedersen et al. 2016). This is a huge undertaking and must be considered from the very beginning of a project.

2.4.4 Loading time

Loading time refers to how long the game takes to load. This includes the first load when the game is opened, and loading that happens during runtime, for example between stages.

While not usually a major issue, having extremely long loading times or having loading screens appear far too often can negatively affect the user experience. To reduce the length of loading screens, consider splitting up the work done during them. This can mean preloading assets beforehand, to reduce the number of objects that need to be loaded during the screen, or reducing the complexity of loaded scenes.

In an open world game, where a lot of objects need to be loaded during runtime, a method of recycling or streaming assets can be implemented (Kjems, Pedersen et al. 2016). In *Inside*, the 2D adventure game by the studio Playdead, a small part of every frame is dedicated to loading and unloading assets. This allows the whole four-hour experience to be played through with just a single initial loading screen.

2.4.5 Memory usage

The memory needed by a game is split into two categories here, RAM memory and VRAM memory. Either or both might become a bottleneck on a project running with unoptimized or simply with far too many assets.

RAM memory stores everything needed by a game during runtime. To reduce the amount of RAM needed by your project, consider reducing the complexity of objects and the resolution of textures, or simply have less unique assets in the project (Echterhoff, Riber 2013).

VRAM memory is the storage used to store textures and models drawn by the graphics card. When there is not enough memory for the graphics card to use, stuttering may occur. This is caused by the component loading and unloading assets from RAM memory (Bavoil

2015). The same trick as with RAM works here. Reducing the amount of different assets and the complexity of those assets helps to reduce the amount of VRAM necessary for the project to run smoothly.

3 Profiling Unity

The first step of every optimization process should be profiling (San Filippo 2015). Profiling means reading the performance data of the project and finding out where the performance issues lie. From this data, bottlenecks can be found and the optimization process monitored.

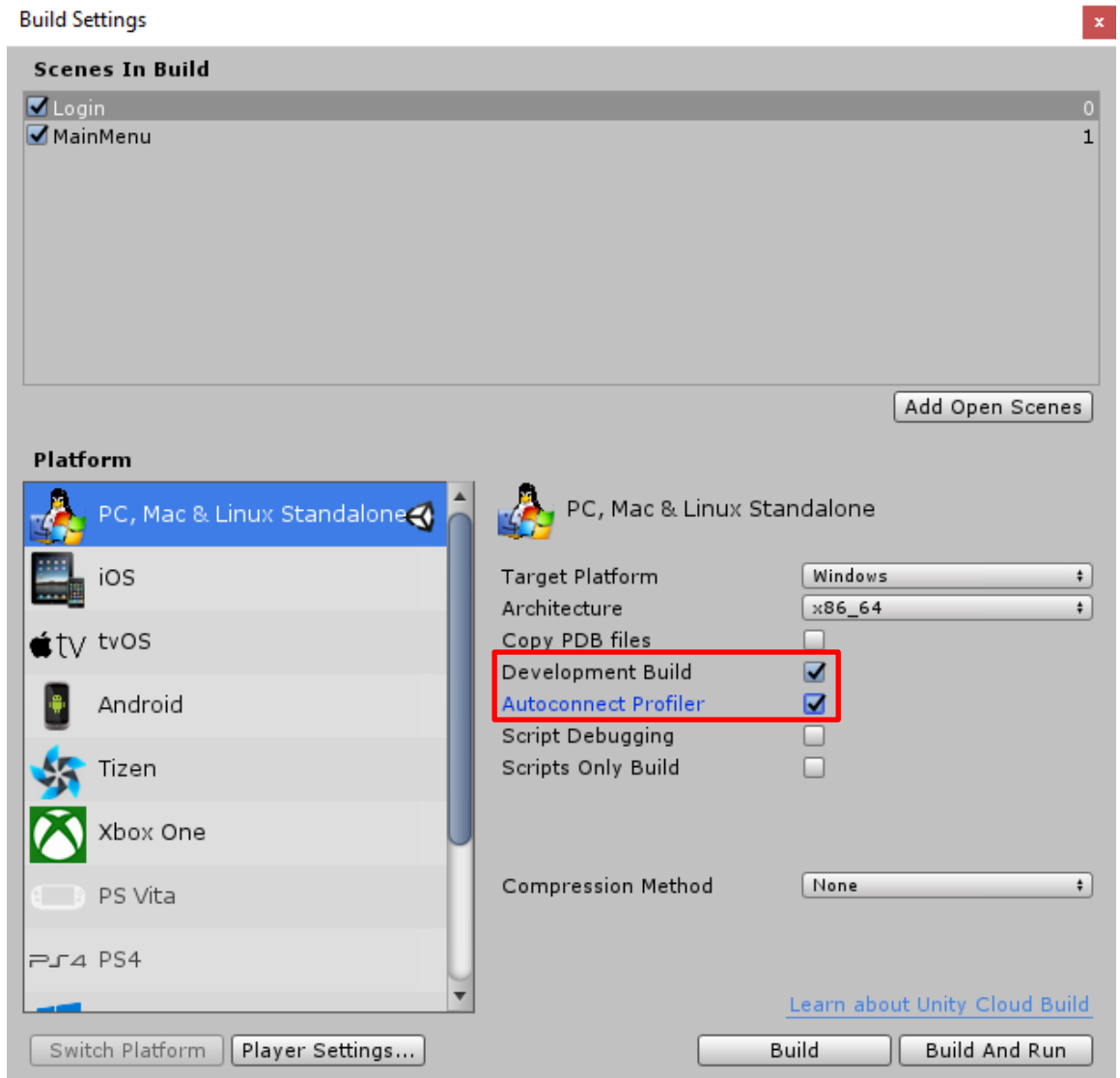
Unity provides three different tools for profiling; The Unity Profiler, the Frame Debugger and the Memory Profiler. This chapter goes through each of these tools, analyses them and gives information on how to use them.

3.1 Connecting the Profiler

The Unity Profiler and the Frame Debugger can both be used to profile a project currently running on the same editor. However, the editor affects the performance of the project and the profiling information can be inaccurate. For this reason, if accurate profiling data is needed, a separate build should be created for profiling purposes.

3.1.1 Making a build for profiling

To have a build for the profiler to connect to, simply use the settings “Development build” and “Autoconnect profiler” while building (Picture 2). This allows the editor to find the build and for the profiler to connect to it.



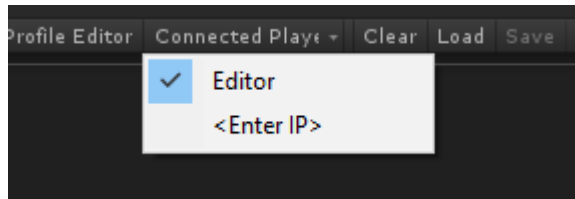
Picture 2. Making a development build

3.1.2 Profiling external devices

To get accurate profiling data, the project should be profiled on a device the end user will use the final product on. Profiling a mobile game on a powerful PC gives little insight on how a user will experience the game.

The easiest way to connect the profiler to an external device is to have the two devices share the same network. If there is a device running a development build of the project, it should be selectable in the “Connected Player” dropdown menu of the profiler (Picture 3). Once the correct device has been selected, the profiler should automatically start profiling it. Profiling on an external device still affects the performance, but the effect is less than

when profiling on the same device. Mobile devices can also be profiled through an USB cord.



Picture 3. The connected player dropdown menu

3.1.3 Profiling mobile devices

Modern mobile devices are powerful but often lack an active cooling system. Because of this the heat of the device affects its performance drastically (Hawkins 2015).

On mobile devices, the performance of a game can be directly compared to the heat level of the device. When the device is just turned on and the application is launched, the performance is at its best. Once the device has warmed up the performance starts to fall as the device tries to cool itself. And again, once the device has cooled down enough, performance is improved (Hawkins 2015).

To effectively profile a mobile device, it should be warmed up first. After the build has run for 10 or so minutes, and the device has warmed up properly, the profiling information becomes more accurate and more applicable to an actual user experience.

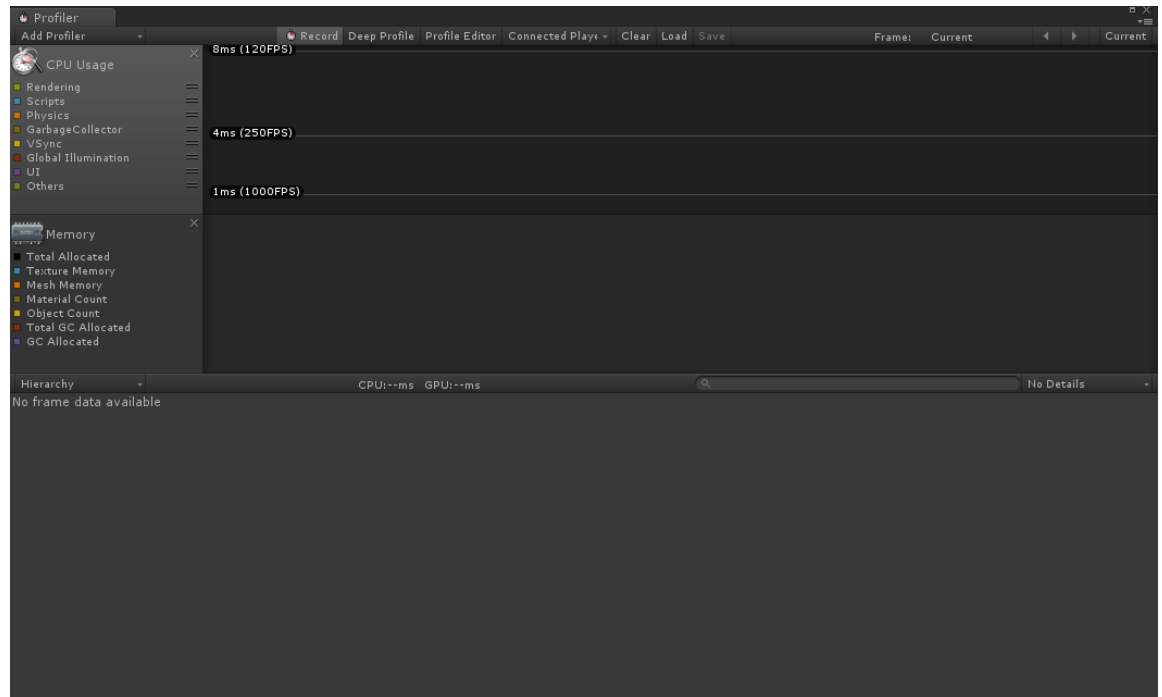
3.2 The Unity Profiler

The Unity profiler is the main tool for profiling Unity projects. It provides a plethora of information on the different performance areas of a project, such as CPU and GPU time, the rendering process and memory usage (Introduction to the Profiler 2013).

This tool is the beginning of every optimization process on Unity. From information provided by it, bottlenecks are found and from these bottlenecks the first optimization tasks are created. Learning to use it helps to understand the innerworkings of the engine and the project at hand.

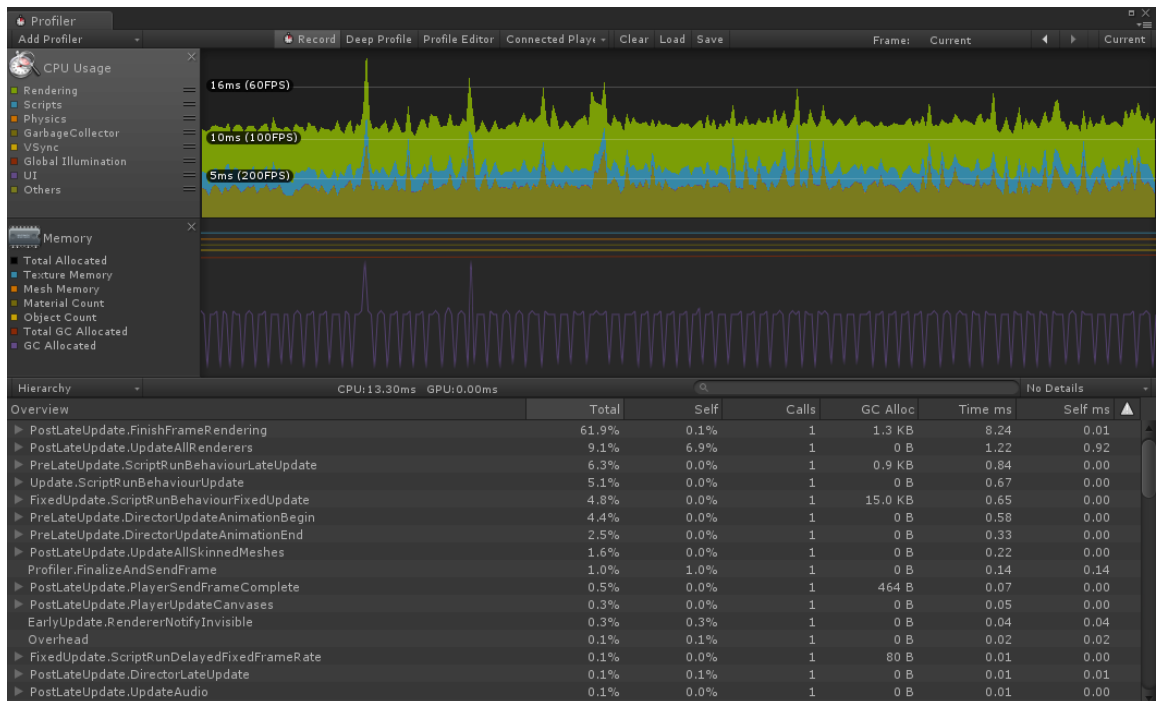
3.2.1 Using the Profiler

The profiler window can be opened through the “Window” tab of the editor, or by pressing the shortcut CTRL+7 (Picture 4).



Picture 4. The profiling window

Once the profiler has been connected to a running build and the record button has been pressed, profiling information starts pouring in. On the left side of the window the different profiling categories can be seen, and more can be added from the “Add Profiler” dropdown menu. Out of these categories the most relevant ones are CPU usage, GPU usage, rendering and memory usage.



Picture 5. Profiling information

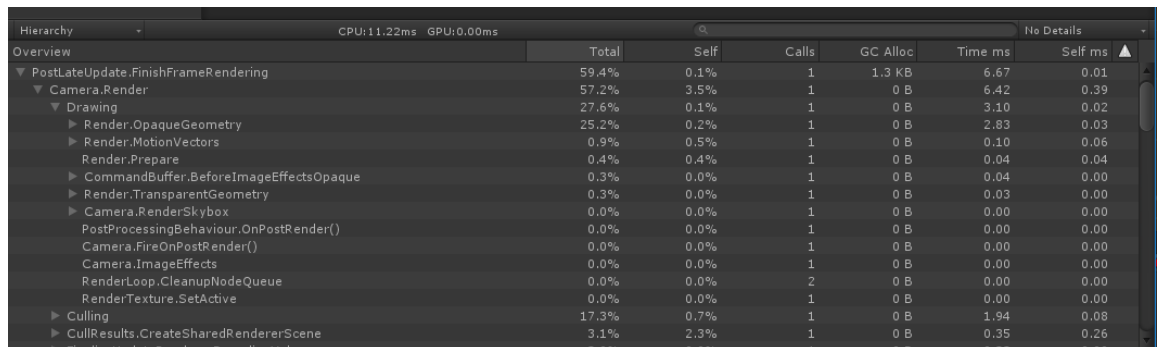
When profiling, information on each active category appears from the right side of the window (Picture 5). Each slice of the information presents a single frame. On the CPU usage category, the time the frame takes to finish is presented by the height of the slice. These slices can be cycled through by pressing the arrow keys on the window, or by clicking a slice with the mouse. Information on the selected category and frame appears under the window. To stop or pause the profiler simply toggle the Record button.

3.2.2 Reading the Profiler

By clicking the different categories on the profiler window, different information is presented on the lower half. For example, when the CPU category is highlighted the information shown is a list of functions running on the CPU (Picture 5).

Next to the function name different values are shown. The values provide information on the performance of the function in the selected frame. These values are the percentage of the total CPU time spend on this function, the number of times the function is called in the chosen frame, how much garbage the function generated and the time it took to finish the function in milliseconds.

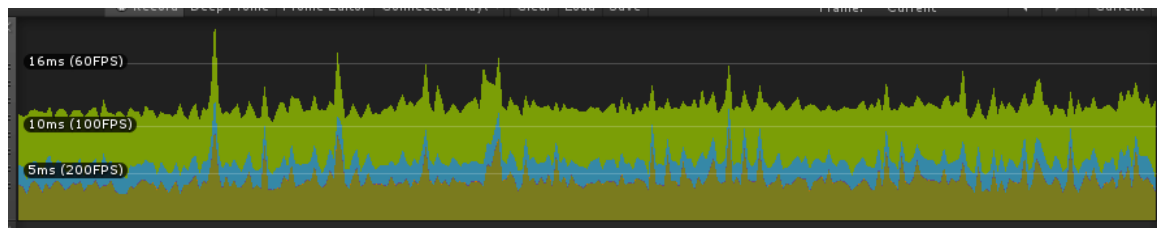
If these functions themselves call other functions, a small arrow is shown next to the name of the function. Clicking this arrow opens a function hierarchy that shows how the full time of the function is split among the functions called by it (Picture 6).



Function	Total	Self	Calls	GC Alloc	Time ms	Self ms
PostLateUpdate.FinishFrameRendering	59.4%	0.1%	1	1.3 KB	6.67	0.01
Camera.Render	57.2%	3.5%	1	0 B	6.42	0.39
Drawing	27.6%	0.1%	1	0 B	3.10	0.02
Render.OpaqueGeometry	25.2%	0.2%	1	0 B	2.83	0.03
Render.MotionVectors	0.9%	0.5%	1	0 B	0.10	0.06
Render.Prepare	0.4%	0.4%	1	0 B	0.04	0.04
CommandBuffer.BeforeImageEffectsOpaque	0.3%	0.0%	1	0 B	0.04	0.00
Render.TransparentGeometry	0.3%	0.0%	1	0 B	0.03	0.00
Camera.RenderSkybox	0.0%	0.0%	1	0 B	0.00	0.00
PostProcessingBehaviour.OnPostRender()	0.0%	0.0%	1	0 B	0.00	0.00
Camera.FireOnPostRender()	0.0%	0.0%	1	0 B	0.00	0.00
Camera.ImageEffects	0.0%	0.0%	1	0 B	0.00	0.00
RenderLoop.CleanupNodeQueue	0.0%	0.0%	2	0 B	0.00	0.00
RenderTexture.SetActive	0.0%	0.0%	1	0 B	0.00	0.00
Culling	17.3%	0.7%	1	0 B	1.94	0.08
CullResults.CreateSharedRenderScene	3.1%	2.3%	1	0 B	0.35	0.26

Picture 6. Function hierarchy

The height of the lines shown on the upper half of the window tells how long that frame took to finish (Picture 7). If these lines have noticeable spikes in them, there has been a framerate drop. Hunting down what function caused the spike and optimizing it is a way of making the framerate and gameplay experience smoother.



Picture 7. CPU usage graph

3.3 The Frame Debugger

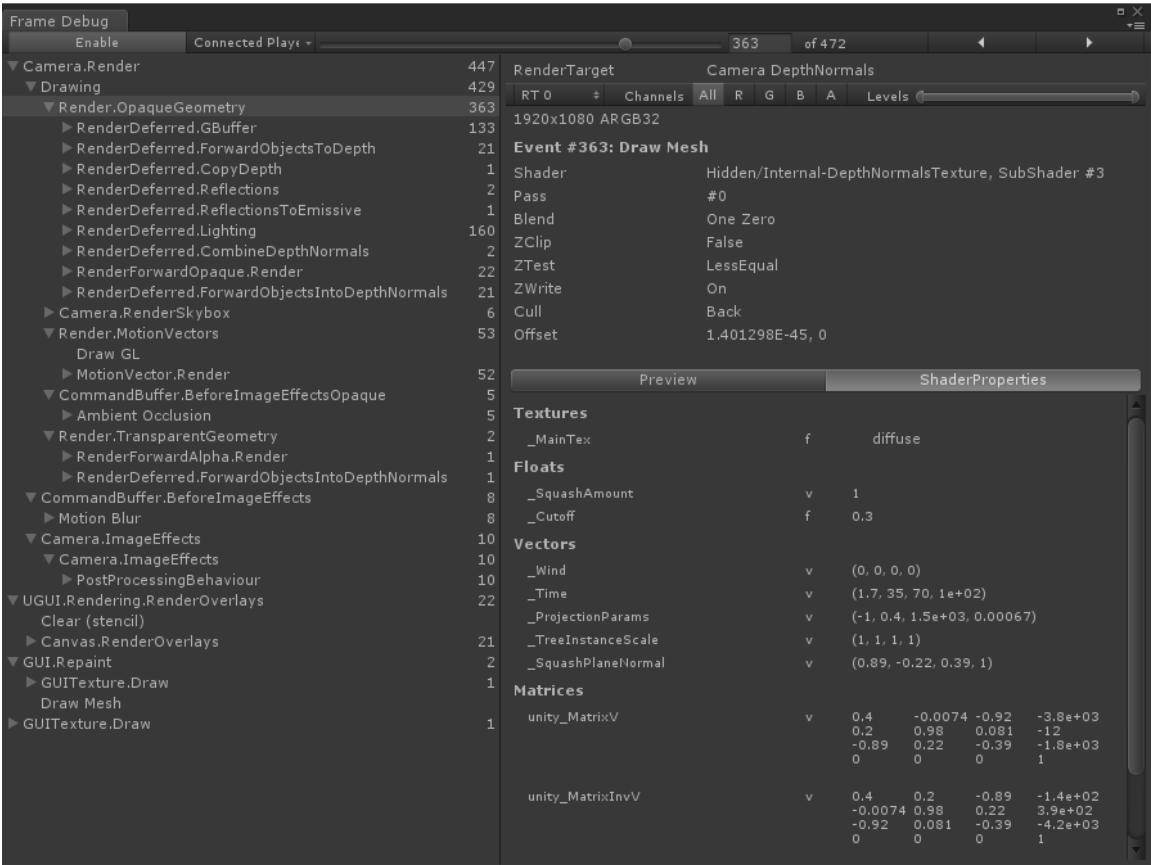
To gain information on what the GPU spends its drawcalls on, the frame debugger can be used. By using the tool, more insight on the rendering process can be gained. It also is used to find the areas where the most drawcalls are spend, and where optimization is required (Frame Debugger 2015).

3.3.1 Using the Frame Debugger

The frame debugger window can be opened through the “Window” tab of the editor.

The debugger is connected the same way as the profiler, by choosing the right player under the “Connected Player” dropdown menu. The frame debugger can easily be used either inside the editor, debugging the gameplay window, or on a separate device. Debugging a build on the same device as the editor can cause some issues as the window is frozen by the debugger.

Once the frame debugger has been activated, it freezes the profiled build and collects data on the frozen frame. Once the data has been collected, information on every drawcall used to render that frame appears on the left side of the window. The drawcalls are organized into hierarchies such as drawing and image effects. Next to the hierarchy name is shown the amount of drawcalls it took to finish rendering that area.



Picture 8. Frame debugger window

3.3.2 Reading the Frame Debugger

By cycling through the drawcalls on the debugger window, changes can be seen on the frozen frame of the profiled build as the debugger shows what was drawn by each call

(Picture 8). By clicking a single drawcall, more information on that specific call appears on the right side of the window.

Cycling through the drawcalls and watching the frozen screen change can give enough of an idea on where optimization is required. For example, if the most of the drawcalls are spend drawing a single object or a character in the scene, work on that object might be required.

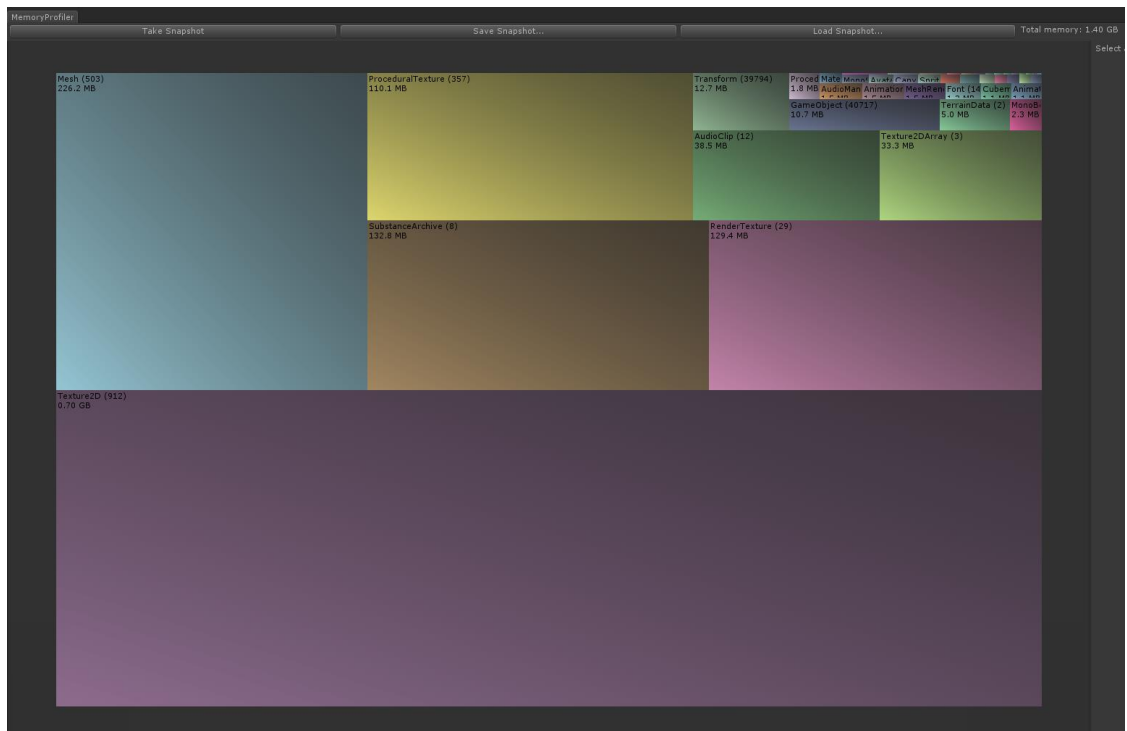
3.4 The Memory Profiler

The Memory Profiler is an external Unity project that can be downloaded from Unity's BitBucket site. The Memory Profiler provides more accurate information on the memory usage of the profiled project. It takes a snapshot of the current memory usage and divides the memory by object type (Harness, Dundore 2016).

3.4.1 Using the Memory Profiler

The memory profiler is a separate project instead of being a part of the engine. To use it, a build of the profiled project is needed. Once the build is running and the memory profiler project is open in the editor, the profiler must be connected to the build. Once connected and information starts to appear on the profiler, from the editor the "Memory Profiler" window needs to be opened. This window can be found under the "Window" tab.

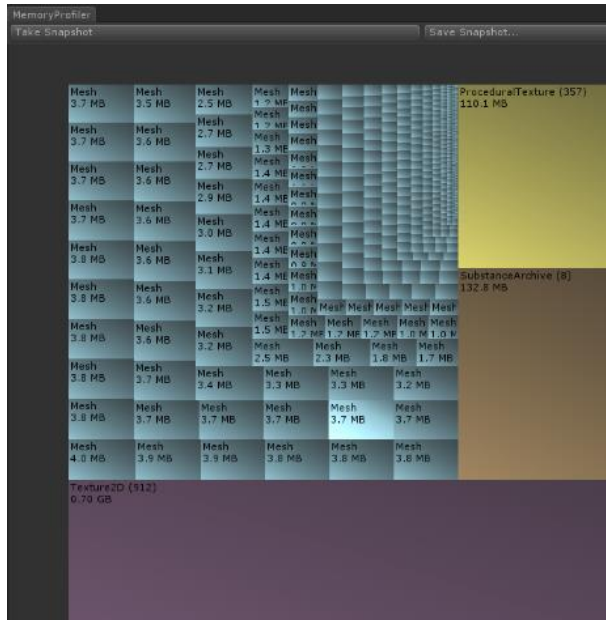
Once the window has been opened, the "Snapshot" button can be pressed. This usually freezes the build for a moment, as information on the memory usage is being collected. This step can require some patience as collecting the memory and building the snapshot can be a lengthy process. Once the snapshot has been build, it shows up on the memory profiler window and the profiled build is no longer needed (Picture 9).



Picture 9. Memory profiler snapshot

3.4.2 Reading the Memory Profiler

The snapshot represents the whole memory usage of the project. Different colored squares indicate memory taken by different object types, for example textures or meshes (Picture 9). By clicking a square, it is divided further into smaller squares representing individual objects (Picture 10). More information on these individual objects can be seen by clicking them, and even a reference on what is using the object can be seen.



Picture 10. Memory profiler area split

Just by looking at the memory usage snapshot split up by object type tells which objects use the most memory. Tracking down the individual big spenders and working on those can quickly bring down the memory used by the project. Especially with the textures, just changing the import settings in the Unity editor goes a long way.

3.5 Benchmarking

Benchmarking refers to collecting data from a project to measure its performance. This is different from profiling in that the benchmarking data should be as close to the performance of the final product as possible. Because of this the data should not be collected by using the Unity Profiling tool, since just running the tool has an impact on the performance.

Benchmarks should be taken on the same device as the final product is meant to run on (Turner, Schell 2017). A single device can be good enough, but for getting a lot of data, multiple devices of different performance capabilities should be benchmarked on. This way more information on the performance of low- mid- and high-tier devices can be gained.

When benchmarking the framerate of the project, a separate script can be used to track the framerate instead of using the profiling tool. To keep the framerate data accurate between benchmarks, the gameplay should be reproduced as accurately as possible

(Kjems, Pedersen et al. 2016). The best way to handle this is to have a script that plays the game the same way every time. When the gameplay is identical between benchmarks, they can be compared to each other and the optimization process can easily be monitored.

4 Optimizing Unity

Every project is unique and has its own unique issues. However, there is a plethora of well-known performance related problems associated with the engine itself. This chapter goes through several of the known performance issues in Unity and provides information on avoiding them.

4.1 Unity UI

The most common optimization issue with new Unity projects is the UI system (Dundore 2016). The basic element of the Unity UI is a canvas. These canvases then house all other elements of the UI, be it text or images. The issue with these canvases is that when a single thing on a canvas changes, the whole canvas is marked dirty and needs to be redrawn.

Often with developers who haven't had much experience with Unity, all the UI in the game is set up under a single canvas (Dundore 2016). This can mean that a complex menu system, even while hidden, needs to be redrawn every frame because it shares the canvas with a timer.

Luckily there is a simple way to approach this issue, multiple canvases. The common method is to split UI elements to separate canvases by update frequency (Dundore 2016). Elements that are updated regularly, like timers, under a single canvas and static elements under another one. This fixes the issue of having to update every single UI element every frame.

UI is a bottleneck especially on mobile devices. Transparent graphics are not handled well by mobile graphics cards, and all the UI elements in Unity are considered transparent (Dundore 2016). For this reason, splitting elements between multiple canvases can have a huge effect on the performance of the project.

4.2 Memory management

In Unity, the memory management is done automatically. Unity is fast and efficient at allocating memory, but deallocating it afterwards often causes a huge framerate drop

(Echterhoff, Riber 2013). These drops are especially noticeable in fast paced games, where stable framerate is important.

Many Unity's own functions generate garbage, making it almost impossible to eliminate the garbage collection spikes, without altering the source code of the engine (Kjems, Pedersen et al. 2016). Following are listed several methods of circling around this garbage collection issue.

4.2.1 Garbage collector

All memory deallocation in Unity is done by the garbage collector (Echterhoff, Riber 2013). When enough memory has been allocated, the garbage collector automatically clears unused objects from the memory (Echterhoff, Riber 2013). This is referred to as running the garbage collector, and always causes a framerate drop.

There are multiple ways to lessen the performance impact and the frequency of running the garbage collector. The simplest is to generate less garbage during runtime (Echterhoff, Riber 2013). When profiling the CPU, the amount of garbage generated by each function can be seen. Then altering those functions to generate less garbage, for example by caching, is a way to lessen the frequency of these garbage collector runs.

4.2.2 Manual garbage collection

One way to dodge the issue is to manually run the garbage collection when the gameplay is less hectic, and the framerate drop won't affect the user experience. For example, during loading screens or during more peaceful periods of gameplay (San Filippo 2015).

In the video game Race the Sun, garbage collection spikes were a major issue. The gameplay in Race the Sun consists of longer periods of active gameplay with a shorter passive moment between them. The way this issue was beaten was to run the garbage collection on those passive moments of gameplay. This still caused a noticeable framerate drop, but at a time where it did not negatively affect the gameplay (San Filippo 2015).

4.2.3 Object pooling

To prevent garbage collector issues in games with a lot of spawning and destroying objects, a method called object pooling can be used (Mignano). Object pooling refers to creating all necessary objects beforehand and disabling / enabling them when necessary, instead of instantiating and destroying objects during runtime.

This is often done by having an array containing disabled objects that are often used, for example bullets or units in a strategy game. When a gun is fired, instead of spawning a new bullet and allocating memory for it, we take a game object from the list, move it to the right position and activate it. When the bullet collides with something, instead of destroying it, we disable it and add it back to our list. This way we can have a lot of objects appear and disappear in game, without allocating any memory and causing issues with the garbage collector (Echterhoff, Riber 2013).

These objects can also be spawned beforehand during a loading screen and kept hidden until needed. This way they won't cause performance issues when spawned during gameplay.

4.2.4 Pre-allocating memory

Unity pre-allocates a minimum amount of memory on startup. This is called the heap. When the heap is filled, new memory must be allocated and this can be performance intensive (Echterhoff, Riber 2013).

One way to avoid the issue is to manually allocate a large amount of memory on startup and then empty that memory. This amount should be close to the measured maximum amount of memory needed by the program. This forces Unity to expand the heap on startup and removes the performance issues caused by expanding the heap during runtime (San Filippo 2015).

4.3 Reducing drawcalls

A drawcall is a function performed on the GPU used to draw the screen every frame (Frame Debugger 2015). Modern 3D games with complex graphical assets and effects may need thousands of drawcalls to render the screen each frame.

If a project is GPU bound, reducing the number of drawcalls is a simple method to lessen the workload of the GPU. Several different approaches can be taken to reduce the number of calls needed for rendering a frame. The simplest way is to remove objects drawn (Echterhoff, Riber 2013). This can be done by removing objects from the scene, but this is not always an option, or by using the culling systems provided by Unity (Echterhoff, Riber 2013).

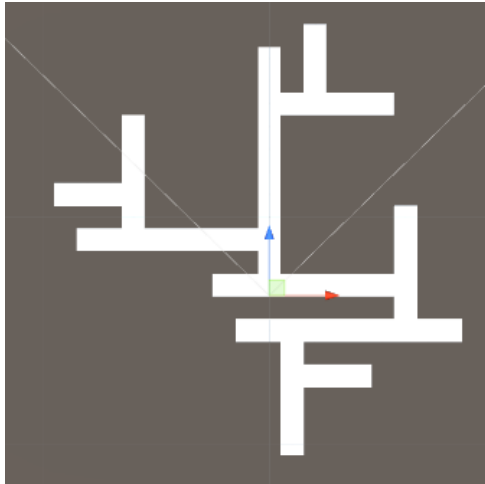
4.3.1 Static game objects and batching

Game objects can be marked as static in the editor. This mark tells the engine that the marked object will never move. This allows the engine to use a method called batching to render it (Echterhoff, Riber 2013).

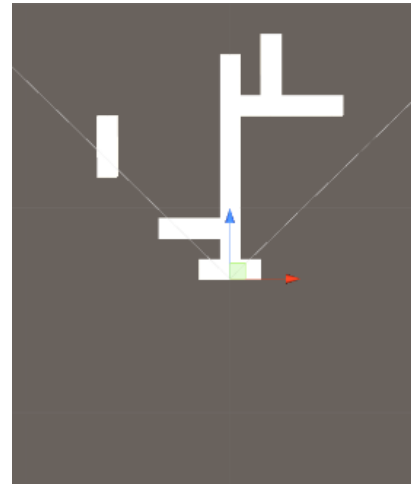
Batching happens when the engine draws multiple objects on a single drawcall. For batching to work the objects must be marked as static and they must share the same material (Echterhoff, Riber 2013). By using the frame debugger, you can see when batching has been used to reduce the number of drawcalls. The debugger will also tell you why a drawcall was not rendered in the same batch as the previous one.

4.3.2 Frustum culling

Frustum culling removes objects outside the camera view from the rendering process (Picture 12), saving draw calls without affecting the user experience (Echterhoff, Riber 2013). In Unity, this is an automated process and is always on, requiring no setup from the developer.



Picture 11. Maze level

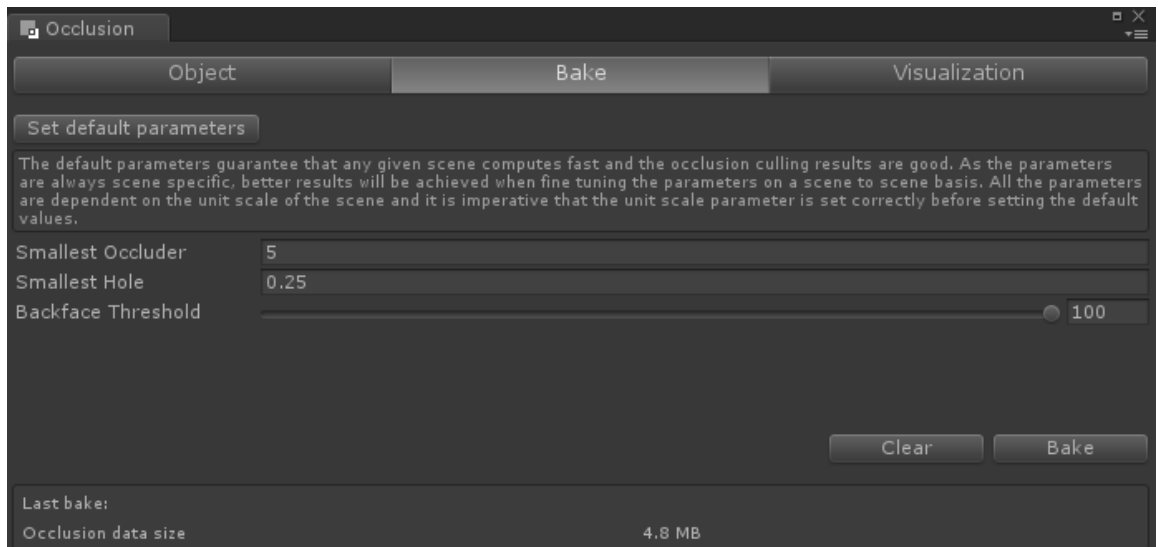


Picture 12. Maze with Frustum culling

4.3.3 Occlusion culling

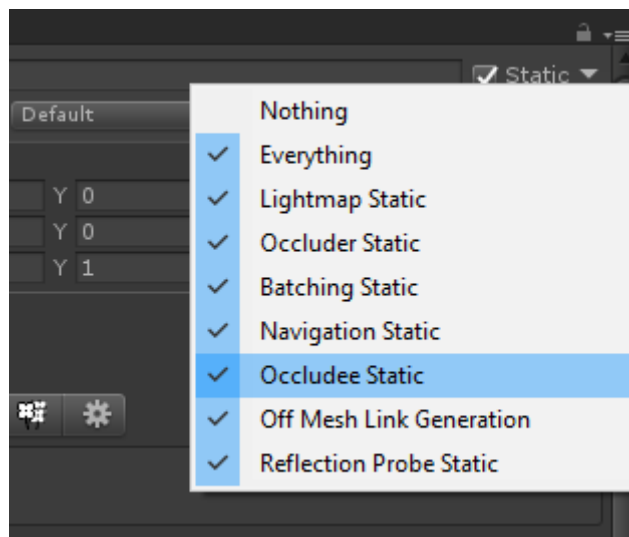
Occlusion culling works alongside frustum culling as an additional method of reducing draw calls. While frustum culling hides objects outside the view, occlusion culling aims to minimize draw calls in view (Picture 16). This is achieved by removing objects hidden behind other objects from the rendering process (Wesolowski 2014).

Unlike frustum culling, an automated process, the occlusion culling must be set up and baked beforehand in the editor (Unity Manual: Occlusion Culling). For achieving this Unity has provided an object called occlusion culling area, which can be created from the occlusion window (Picture 13). The occlusion culling window can be found under the “Window” tab of the editor. These objects need to be placed in the world before the data can be baked. This can be done by hand or through scripts.



Picture 13. Occlusion culling window

Game objects with the tags Occluder Static and Occludee Static are included in the occlusion culling process. These are on by default in static game objects (Picture 14). When an object marked with the occluder tag hides an object marked with the occludee tag behind it from camera view, the occludee object is removed from the rendering process.



Picture 14. Occlusion culling tags

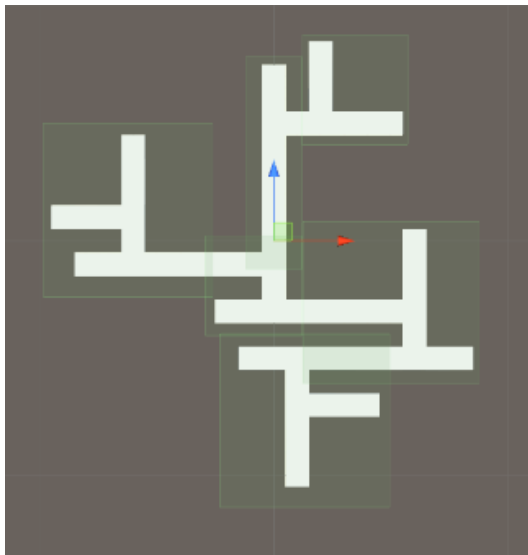
The occlusion areas are to be set up such that the camera will never wander outside them (Picture 15). Should the camera go outside the occlusion culling areas, no occlusion culling will be applied, causing errors such as invisible objects and holes in the terrain.

Once the occlusion areas have been set up, occlusion culling can be baked. This is done from the occlusion window. Once completed the effects of occlusion culling can be viewed

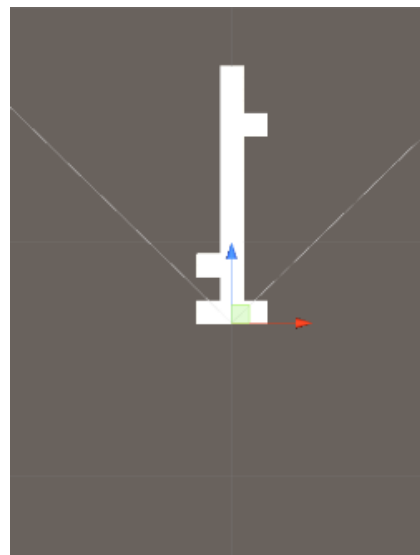
by using the visualize tab on the scene window. This allows for debugging by moving the camera and observing the changes in editor.

Should pop-in of objects and other errors occur, there are few ways to approach the issue. The values provided in the occlusion window can be tweaked, affecting the precision of the culling, the size of the baked file and the time it takes to finish the baking process.

Other way of adjusting the culling precision is to change the size and frequency of occlusion areas. Multiple small areas can be more precise compared to a single larger area, but can bring up the baking time and the size of the baked occlusion data file. If there are still objects that are not correctly culled, they can be taken out of the occlusion process entirely by removing the Occludee Static tag from the game object.



Picture 15. Maze with occlusion areas



Picture 16. Maze with occlusion culling

4.3.4 LOD levels on 3D objects

LOD or level of detail is a system used on 3D objects to reduce their impact on the performance. It functions by replacing the high-quality 3D object with a similar object of lower quality when they get further away from the camera (Wesolowski 2014). Often the 3D object is replaced further with a 2D render of it and even removed completely.

LOD levels can be set up by adding the LOD component to the game object (Unity Manual: Level of Detail (LOD)). From the component the amount of levels and the level changing distance can be modified. The way Unity determines when the LOD level should be changed, is how much of the object is on the screen at a given time. This can cause weird

behavior in objects partially hidden or underground, for example rocks. This method of reducing drawcalls causes more work on the artists end since the lower quality objects need to be made by someone.

4.4 Optimizing scripts

This chapter provides information on methods of making the scripts more optimal. There are multiple known ways of tackling the common performance issues on the programming side. Just remember that the most optimal code is the one that doesn't have to be run.

4.4.1 Preventing spikes

As mentioned before spikes are caused by difficult and complex operations performed during a single frame. These issues can be prevented by splitting the operation along multiple frames. They can also be performed ahead of time or when the gameplay is less intensive (Kjems, Pedersen et al. 2016).

4.4.2 Caching

Caching refers to storing something that is often needed as a variable instead of calculating or fetching it whenever it is needed. For example, this can mean storing the camera as a variable in a script instead of calling `Camera.Main` function whenever the camera is needed.

These Unity API functions such as `Camera.Main` or `gameobject.transform` all cause a little bit of overhead (Echterhoff, Riber 2013). `Camera.Main` is a shortcut for finding a camera object from the scene with the tag `Main`. This value is never stored so every time the function is called, a search operation is performed. All this overhead can be eliminated by storing a reference to the object as a variable after the first call of the function (Dundore 2017).

5 Optimizing the Virtual Frontiers project

This chapter gives information on a project I worked on during my internship in Virtual Frontiers. The task given was to optimize a company project while porting it on Mac devices.

This chapter will go through some of the unique challenges, that came up during the task, and the solutions reached for those challenges. A lot of trial and error was needed to reach the solutions and it will be shown in the chapter.

5.1 Benchmarking

To gain proper performance information like what the end user would experience, benchmarking on the Mac devices was needed. To gain accurate data from multiple different devices and quality levels, an automatic benchmarking system was created.

The gameplay of this project consists of multiple linear tracks, that can take a lot of time to finish. This would mean two things; The gameplay would not be too difficult to automate and that benchmarking through the whole project would take multiple hours.

A script was set up such that we had a single “hub” scene from which the other scenes were launched. The script would start the project on the lowest quality setting in the hub scene. From this scene it would open and play through the gameplay scenes, always returning to the hub after a scene was completed. After finishing all the gameplay scenes, the quality level would be increased and playing through the scenes started again.

The automatic system would also collect performance information while running. This information was then stored on text files created by the script. These files were saved on the hard drive of the device and organized by scene and quality level. This script would often be left running on two or more Mac laptops over the night and fresh data could be gathered on the next morning.

5.2 Occlusion culling

Thanks to the game running on a single linear track, the occlusion culling areas could be set up automatically. A script was made, that took the track and automatically created the occlusion culling areas along it. This removed the need for human touch from the creation process of these areas, and removed that workload entirely. Whenever changes would be made to the scenes, the areas could be set up automatically and the occlusion culling data baked easily.

5.3 The UI

As is often the case, the UI was shown to be a bottleneck in this project as well. Like mobile devices, the Mac laptops had an issue with rendering transparent graphics like the whole UI system.

The simple solution worked the best and the UI was split among multiple different canvases by update frequency. This seemed to fix the issue and the task was considered done.

After more optimization work had been done and the overall performance of the project increased, the UI once again became a bottleneck. The assets of the project included road signs that had the text on them done by using a UI canvas. This was found to be a bad idea, since UI elements that move and resize in 3D space as the player moves caused an even larger performance issue.

This was fixed by dividing the UI further on even more canvases and flattening the hierarchy of those canvases. The canvases on the signs were replaced with transparent textures with writing on them, and the performance impact of the UI was once again brought down.

5.4 Terrain issues

From the very beginning of the optimization process, the terrain system of Unity proved a huge bottleneck on Mac devices. A LOD system used by the Unity's default terrain system

caused a lot of computations to be done on the GPU, which was one of the weaker components on Macs. By disabling the terrains, a large performance gain was achieved, which caused me to think of ways to replace the default terrain system.

The first idea was to replace the terrains with meshes. A third-party component from the Unity's asset store was used to generate meshes from the default terrains. These meshes eliminated the need for LOD computations during runtime, but did cause the project to use a lot more memory. Another issue was caused by the mesh terrains however, and it was the foliage system.

The default foliage and tree system of Unity is integrated into the terrain system. By replacing the terrains with meshes we lost the trees and plants used in the project. The first idea to fix this was to have an invisible default terrain just under the mesh terrain to run the foliage system on. This terrain would only contain the heightmap for the correct placement of the foliage and the foliage system itself. This was however deemed dodgy and a better system was needed.

The second option was to try one of the multiple foliage systems found in the asset store. After testing a few assets, a working option was found and that system was used alongside the mesh terrains to have a terrain comparable to the default one with far better performance.

However even this solution wasn't perfect and multiple issues were caused by the new foliage system. Once the Unity version was updated, and further benchmarking was done on the terrain systems, not much performance was gained by using our system compared to the default one. This caused us to backtrack into using the default terrain and foliage systems once more, losing a little performance but gaining back the lower memory usage. The performance lost was quickly gained back by opting to always use the lowest LOD level of the terrain, losing some of the fidelity but removing the need for runtime calculations. The other step to greatly increase the performance was to replace most of the trees in scenes with 2D images, lessening the workload of the foliage system without much affecting the user experience.

5.5 Terrain streaming

A simple issue in some of the scenes of the project was the sheer size of them. Just having a lot of objects, especially terrains, in a single scene caused the performance of that scene go down drastically.

Multiple approaches were tried to limit the number of objects and the size of the scene. Soon it was clear that a streaming system would be necessary. The first one tried was a system to stream scenes; The initial scene was split unto multiple smaller scenes and they were loaded and unloaded when necessary. This proved to be nonfunctional in our case since the loading and unloading would always cause a framerate spike that could freeze the game for multiple seconds.

The second system tried was just streaming the terrains inside the scene. Instead of having over 30 terrains always active in the scene, only nine terrains around the player character would be active at once. For the rest of the terrains a boundary was set up that would trigger when the player would move inside it. When a boundary was triggered, the terrains behind the player would be removed and new terrains would be spawned in front of the character.

The system was working, and just by having less active terrains the performance was improved, but the framerate spikes would still occur when terrains were loaded and unloaded. The framerate spikes were then eliminated completely by removing the spawning and deleting of objects from the system. The next iteration of the streaming system worked by moving the already existing terrains in front of the player when necessary and replacing the data of the moved terrain by a new one. This way no spawning was necessary, and instead the same nine terrains were being recycled through the whole scene.

6 Conclusion

Every project is unique and the performance issues found in them are unique as well. Learning the usage of the tools provided helps immensely in finding and understanding these issues.

The optimization process should be started in a projects lifetime as soon as there is something to optimize. This should be done before there are too much to change if big changes are needed. However, over-optimization can also be an issue. Remember to not spend time optimizing what, in the end, doesn't provide much performance gain. If the performance issue can't be measured, it is not an issue.

When starting a project, take some time to consider some baseline practices for the assets created for the project. Do you really need those 2K textures for your mobile game? Or does it really need these complicated calculations done every frame? Changes to an already existing asset- or codebase can be tedious and take a long time to do. Doing them before the project has grown too large to handle can save a lot of time.

References

1. Bavoil L. Are you running out of video memory? detecting video-memory overcommitment using GPUView. <https://developer.nvidia.com/content/are-you-running-out-video-memory-detecting-video-memory-overcommitment-using-gpuview>. Updated 2015.
2. Hawkins K. Uncover your game's power and performance profile. <https://www.youtube.com/watch?v=6Ra3ouahAzM>. Updated 2015.
3. Unity manual: Occlusion culling. <https://docs.unity3d.com/Manual/OcclusionCulling.html>.
4. Turner K, Schell M. Performance optimization for beginners. <https://www.youtube.com/watch?v=1e5WY2qf600>. Updated 2017.
5. Unity manual: Level of detail (LOD). <https://docs.unity3d.com/Manual/LevelOfDetail.html>.
6. Unity. Frame debugger. <https://unity3d.com/learn/tutorials/topics/graphics/frame-debugger>. Updated 2015.
7. Unity. Introduction to the profiler. <https://unity3d.com/learn/tutorials/topics/interface-essentials/introduction-profiler>. Updated 2013.
8. Mignano M. Object pooling design pattern in unity C#. . . <http://gamedevelopmenttips.com/object-pooling-design-pattern-in-unity-c/>.
9. Gesota R. A simple how to guide for graphics optimization in unity. . 2016. <http://www.theappguruz.com/blog/graphics-optimization-in-unity>.
10. Dilmer V. Unity3D mobile optimization tips. . 2017. <http://dilmer-games.com/blog/2017/10/02/unity3d-mobile-optimization-tips/>.
11. Wesolowski J. How to plan optimizations with unity. . 2014. <https://software.intel.com/en-us/articles/how-to-plan-optimizations-with-unity>.
12. Mignano M. How to optimize and increase performance in unity games. . . <http://gamedevelopmenttips.com/increase-performance-in-unity-games/>.

13. Unity manual: Practical guide to optimization for mobiles. <https://docs.unity3d.com/Manual/MobileOptimizationPracticalGuide.html>.
14. Dundore I. Let's talk optimization. <https://www.youtube.com/watch?v=n-oZa4Fb12U>. Updated 2016.
15. Echterhoff J, Riber KS. Performance optimization tips and tricks for unity. <https://youtu.be/jZ4LL1LqF8>. Updated 2013.
16. Harness M, Dundore I. Optimizing mobile applications. <https://youtu.be/j4YAY36xjwE>. Updated 2016.
17. Harness M, Dundore I. Squeezing unity: Tips for raising performance. <https://www.youtube.com/watch?v=wxitqdx-UI>. Updated 2017.
18. Simonov V. Practical guide to profiling tools in unity. <https://youtu.be/OSIOwJP8Z14>. Updated 2017.
19. Kjems K, Pedersen ER, Madsen ST. Tools, tricks and technologies for reaching stutter free 60 FPS in INSIDE. <https://www.youtube.com/watch?v=mQ2KTRn4BML>. Updated 2016.
20. San Filippo A. Race the sun optimization. <https://www.youtube.com/watch?v=3E946igYpZ4>. Updated 2015.