Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Control Engineering and Information Technology

Menyhárt Bence

GAME ENGINE SPECIFIC OPTIMIZATION TECHNIQUES FOR UNITY

CONSULTANT

Dr. Magdics Milán

BUDAPEST, 2021

Contents

HALLGATÓI NYILATKOZAT

Alulírott **Menyhárt Bence**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2021. 01. 21.

...…….

Menyhárt Bence

Összefoglaló

Valós idejű alkalmazások fejlesztésekor az alkalmazás teljesítménye a fejlesztés során kulcsfontosságú. Sajnos a fejlesztőktől gyakran hiányzik annak a tudása hogyan is kellene egy-egy teljesítmény csökkenést okozó problémát szakszerűen kezelni, a pontos okot azonosítani. Játékfejlesztés esetén ezeknek a teljesítmény problémáknak az azonosítása és kezelése ráadásul még nehezebb, hiszen egy-egy játék rendszerint több platformra kerül kiadásra valamint több egymástól különböző terület munkáját foglalja össze mint például a művészet, zene vagy történetmesélés.

Ebben a szakdolgozatban megnézzük, hogy a fejlesztési folyamat során hogyan és mikor kell meghatároznuk a cél hardvert és milyen optimalizációs célokat érdemes hozzárendelnünk a projektünkhez. Ezek után elmélyülünk a népszerű játékmotor a Unity működésében, különös tekintettel a szkriptkörnyezet működésére. A szakdolgozat folyamán meghatározzuk, hogy a Unity mely részei teljesítménykritikusak vagyis igényelnek különös figyelmet a fejlesztőktől. Összegyűjtjük milyen típusú eszközök állnak rendelkezésre teljesítményelemzéshez és hogyan kell ezeket az eszközöket szakszerűen használni annak érdekében, hogy teljesítmény teszteket és teljesítmény összehasonlításokat végezzünk. Mindezek pedig azt a célt szolgálják, hogy a célközönségünk minél minőségibb játékélményben részesüljön. A szakdolgozat vége felé megvizsgálunk néhány speciális tudást igénylő optimalizációs technikát, majd elemezzük őket annak érdekében, hogy megértsük mikor és miért teljesítenek jobban egy-egy esetben.

Abstract

When we are developing real-time applications performance is a crucial part of the development process. Unfortunately, the knowledge how to handle performance issues or how to accurately identify the root causes of it are often missing from the developers. In game development, identifying and handling these issues are moreover even harder and a broader topic since a game is usually deployed to many platforms and consists of the work of many fields like art, audio or storytelling.

In this thesis we will learn how and when in the development process we should define our target hardware and what kind of optimization/performance goals we should assign to the project. After this we will deep dive into how the popular real-time development platform Unity works, concentrating especially on its scripting. We will identify which parts of it are performance critical and therefore need special attention from the developers. Afterwards we will gather what type of tools are available for performance analysis in Unity and how to use these tools to properly do benchmarks and performance tests in order to ensure smooth gaming experience for our target audience. At the end of this thesis, we will examine some advanced Unity specific optimization technique and understand why and when they perform better.

1 Introduction

In computer science optimization is a process to modify our system in a way to work more efficiently by running faster on the target hardware and/or using fewer resources. This can be achieved at many levels and at different stages of the development process affecting various stakeholders.
For example, reducing the shadow distance and shadow resolution for objects further away in a slightly tilted top-down game would dramatically improve the CPU performance thus directly affecting our end users. This is an optimization that would be typically made at a later stage in development and have a significant business impact.
On the other hand, making our unit tests, integration tests run faster by optimizing the test environment will not directly affect our end users but our development team. These kinds of optimizations are also important and are typically made throughout the project lifecycle when they reach a certain business impact. Sadly, these types of optimizations tend to be ignored by the management but they can considerably speed up development iterations thus should be always take into consideration.

In game development, optimization is an even more broad and complicated process than in general software engineering. The development pipeline includes the work of artists, writers, audio engineers, different kind of programmers (user-interface, gameplay, graphics etc..), testers and many more. We can clearly see that the stack is far larger than e.g. in web development. If we stick to the shadow distance optimization example above, a programmer would be satisfied with a decrease of 15-30% CPU load by halving the shadow distance, but our environment artist would be disappointed after realizing that outside of a 150 unit range shadows would be culled. So, we the programmers might require to make compromises with our colleagues when applying an optimization to a sub system that affect the area of their expertise.

As the above example tries to illustrate optimization in game development is a truly exciting journey. We can optimize our 3d models, textures, audios, file sizes, the algorithms that drive our AI, the loading times and many more and probably half of this does not even depend on the programmers, although the background knowledge why these are needed is usually within our stack.

To take part in this exciting journey, we will use the popular real-time development platform Unity. We are going to learn how we can squish out as much performance as we can from it, via the glasses of a programmer. Unfortunately, many of the principal and leading engineers in game development are sharing their vast knowledge via blogposts rather than via official research forums so researching in this topic differs from other areas of software engineering in general.

2 Optimization goals

Game developers are always pushing the limits of hardware thus optimization is a crucial part of the development. But how do we decide when we should optimize? What should an optimization goal contain? How to decide if the proposed solution is in fact optimal?

Firstly, every optimization goal starts with an observation. It's usually identified by a single observation like

"The game freezes for seconds when opening the inventory."

After this observation, the project lead forwards the observation to the appropriate team. The commissioned team studies the report then starts to identify where the problem that can cause this freeze is. This is done by profiling and testing. The developers attach a profiling tool to the game so they can get results for CPU, GPU, memory, renderer, audio and storage usage. After analyzing the results, they can then identify which parts of the software cause this undesired effect. The commissioned team writes down these observations and proposes an achievable performance goal, then forwards them to the project lead. After that the project lead prioritizes the optimization goal and according to its priority the optimization will be applied. The optimization goal is fulfilled if the proposed performance goal is achieved. As we can see optimization is not about writing a truly optimal solution but writing a solution that is optimal for the given metric/goal. That's why it is important to consult with the appropriate team before setting up an optimization goal since they have the knowledge to setup an achievable one.

2.1 Recording optimization goals

There are numerous ways to record optimization goals. In agile development they are usually recorded via the level of stories (performance related ones, if the team maintains such category). In a requirement driven development (quite uncommon in game development) they are bounded to performance related requirements. But they can be simply maintained on a dedicated wiki section as well, the choice is in the team's hand.

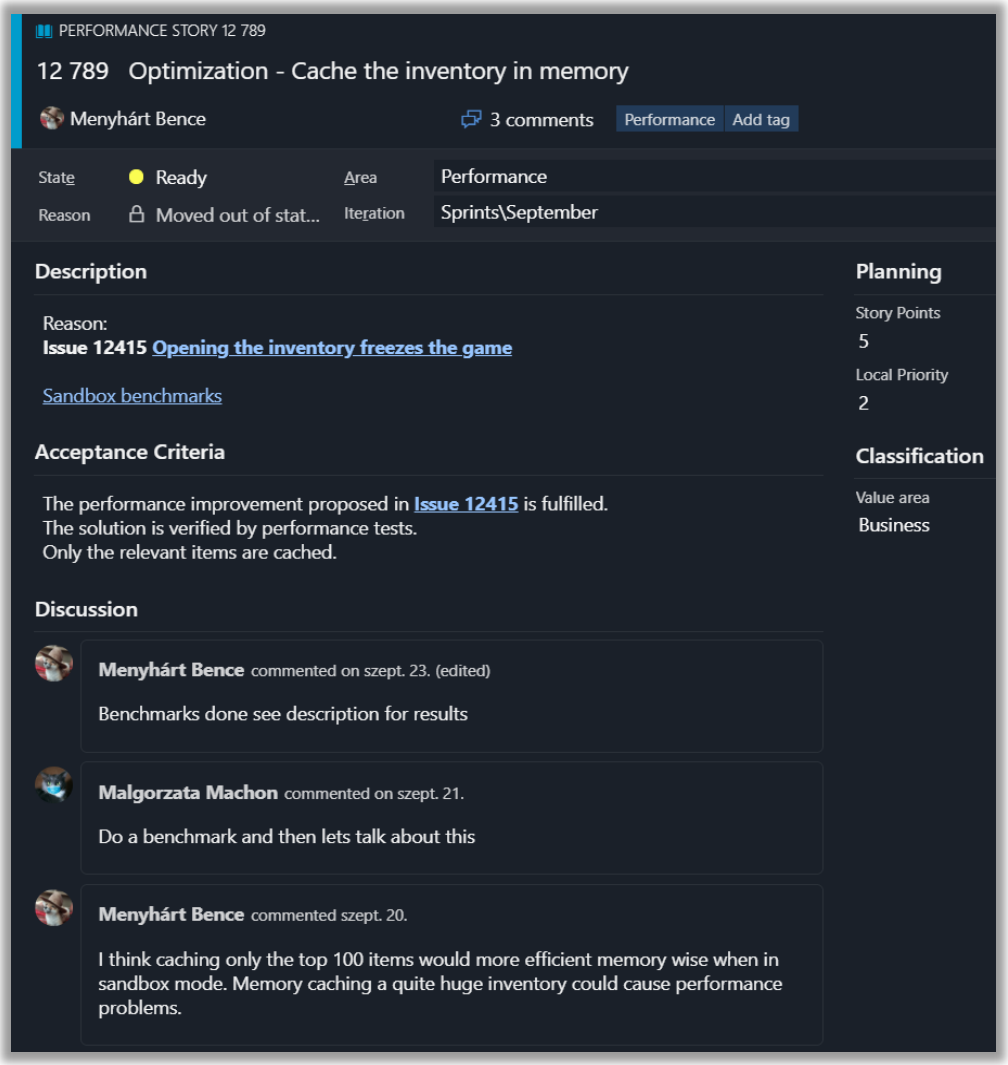

Figure 1: An optimization goal in an agile project using Azure Boards.

We can clearly see the aforementioned four stages in **Figure 1**. Firstly, an observation was made and then an issue was created. The issue got moved to the appropriate team's backlog[1] so it can be investigated and analyzed. After the team finished with investigating, they closed the issue and assigned an achievable optimization goal to the problem (recorded in the issue). The project lead accepted the goal created a follow up story[2] (**Figure 1**) and assigned a priority to it. The Acceptance criteria[3] clearly states that the story can only be marked as verified if the optimization fulfills the goal and it is verified by performance tests.

[1] A list of stories that needs to be done by the team. They are usually ordered by priority.

[2] Represents a feature that needs to be done in agile development

[3] A story cannot be marked as verified till it does not fulfill all the requirements listed in it

2.2 Pre-defined optimization goals

If you have ever searched for an optimization problem you probably heard about the following quote from Donald Knuth

"Premature optimization is the root of all evil."

Sadly, I see this quote quite often misinterpreted and dragged out of context. People of Stackoverflow, Unity Answers and other forums are tend to quote it mindlessly, and forget its original context. They usually connect it with the most of the time right fact that we should implement first then optimize our code (which would be rather implement, measure and optimize anyway), so any thoughts about optimization before the implementation make no sense and actually harmful, at least that's what they suggest. Although the above statement is mostly true, however, when a game concept is born and the programmers have the task to architect the foundation of the game (we are not talking about the engine here but the game's foundation) they should already have a solid grasp where the possible bottlenecks and critical parts can be without writing down a single line of code. Therefore, at the architect phase we should already define some kind of optimization goals depending on our project specific needs and these are far from premature, useless and harmful. And now allow me to put the above quote into its original context

> *Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*
>
> *Donald Knuth, Premature Optimization[1]*

As we can see it's about worrying unnecessarily about non-critical parts of our code but worrying about critical parts is indeed a wise thing to do as the professor suggests. In conclusion one shall not fall into the pitfall of micro optimizations especially at non-critical parts, but identifying critical components and defining optimization goals before the implementation begins is a good strategy.

2.3 Game development pipeline

In this section we are going to take a look at how a game development pipeline generally looks like, and at what stages we should define the aforementioned optimization goals.

Game development pipeline describes the process of making a video game from an idea to a finished product. We throw the basic idea and concept into the start of the pipe, and at the end of it a finished product will flow out. But what are between these two ends? The answer is it depends. AAA studios usually have their own pipelines, and they quite often have different pipelines for their distinct titles. Indies might don't even follow a pipeline (although they usually do so unintentionally), which is a common mistake amongst them. A famous studio like Ubisoft when decides to add one more sequel to one of their leading title, will probably omit steps from the pipeline like prototyping core game mechanics since they already know that the prequels were a hit and the mechanics are great and fit together. On the other hand, an Indie studio should always make a prototype in order to avoid common mistakes like the game is not even that enjoyable like on paper or the desired mechanics simply cannot be executed. So, although different projects and studios might use different pipelines, we can still generalize a rough pipeline that should be kept for every studio and project, and from which we will be able to define when and what type of optimization goals we should specify.

A general game development pipeline contains four major phases, namely Concept, Pre-production, Production and Post-production.

Figure 2: The four major stages of a game development pipeline

Each of these phases contain various topics that need to be "answered", some of them can vary from project to project but some of them are project independent.

2.3.1 Concept

The game concept is basically the rough idea of the game e.g. The First Tree's rough concept was

"Make a game about a fox with minimalistic graphics and deep narrative storytelling"

Figure 3: The cover of The First Tree. [3]

The rough concept as we can see is usually just a few sentences or paragraphs. It roughly describes the game and some key elements.

Once the rough concept is defined, we need to specify things like:

- What are some of the key features?
- Who is the target audience?
- What are the target platforms?
- Are there any competitors?

Even the concept phase is important from an optimization point of view. Since we define our initial target platforms here, we already have a basic idea that what type of limitations we will encounter. E.g. different consoles support different shader models, mobile games should not be computing expensive otherwise the mobile's battery will drain fast. The key features will also give us a hint what type of bottlenecks we might encounter.

If the game concept is something like

"An RTS game with tens of thousands of units where the player can switch to an NPC to control it and fight in third person mode"

we should inform the management that the console ports are not viable, and that we need serious optimizations at AI controllers, character details, and that the target audience will probably be those who have the latest high-end hardware. Although we will have solid idea about what type of pre-defined optimization goals we might need, they are typically collected in the next stage of the pipeline.

2.3.2 Pre-production

Now that the concept is defined, we can start to elaborate in details, and estimate the financial side of the project too. At the end of the phase the team should produce a prototype to verify that the game is indeed enjoyable. In this phase we need to specify things like:

- Define the story in details
- Define the gameplay mechanics
- Estimate the cost of the game
- Estimate the production time and the required human resources
- Define how to monetize the game
- Make a prototype to verify the concept
- Create a Game design document [4].

Although it might sound strange to deal with monetization at such an early stage in development but today's games are not only monetized via how many copies are sold but through IAP[5], advertises and small content DLCs[6].

---

[4] Game design document is a descriptive document of the game. It is maintained and created by the development team in order to guide the team through the development process. The main goal of the document is to describe the game's properties like art style, target audience, characters, story and more.

[5] **I**n-**a**pp **p**urchases is a common way to monetize free to play games

[6] **D**own**l**oadable **c**ontent is an expansion to a game in order to extend it with more playable content.

This is also the key phase for pre-defined optimization goals since this is the phase where we should define them and write them down. We already know the target platforms and the desired gameplay mechanics so the engineers can collect some of the possible optimization goals based on this information. Moreover, as mentioned above, the team should make a prototype to verify the gameplay, which can also be helpful to identify possible future bottlenecks.

When defining pre-defined optimization goals, most of the time we do not provide complete solutions but rather guidelines in order to achieve optimal performance by default on pre-identified critical parts. E.g. if the game heavily depends on AI and physics and one of the main goals is to make the game available to as broad audience as possible, we should assign optimization goals like

- Research path finding solutions for our use cases to ensure smooth movement.
- Setup performance tests for AI and physics common use cases to ensure a stable frame rate.
- Compare the performance of the available physics engines, in order to define which performs better in our use cases.

Unity for example by default has 4 physics engines

- The built-in 3D physics based on an older Nvidia PhysX[7] version.
- The built-in 2D physics based on Box2D[8] engine.
- Unity physics, a complete deterministic rigid body dynamics and spatial query system written entirely in high performance C# using DOTS[9].
- Havok Physics, an implementation of the Havok physics[10] engine for Unity built on top of Unity physics (It requires a special license for Pro users).

Comparing the above-mentioned physics solutions for our use cases will allow us to build upon the most optimal solution thus we won't be limited by the library which we usually unable to modify but change completely.

---

[7] PhysX is a scalable multi-platform game physics solution supporting a wide range of devices, from smartphones to high-end multicore CPUs and GPUs. PhysX is integrated into some of the most popular game engines, including Unreal Engine, Unity3D and Stingray.

[8] Box2D is an open source 2D physics engine written by Erin Catto

[9] **D**ata-**O**riented **T**echnology **S**tack, a solution by Unity to take advantage of modern multicore processors

[10] Havok physics is an industry leading complete physics solution for games and modelling software

Now let's look at an example where we did not set up these pre-defined optimization goals. We are at the middle of the production phase, when we get reports that the default built-in 3D physics' collision detection solution is producing undesired stutters in our fps. After identifying where the exact problem is we conclude that we are unable to solve it without the help of Unity. Sadly, we do not have $100,000 to buy Unity's source code and change the implementation, so we have to find a better alternative physics solution than the built-in one. After benchmarking and measurements, we conclude that Unity's new DOTS based solution will be sufficient for us, however DOTS requires ECS[11] as opposed to our component and object-oriented view. Unfortunately, converting all of our already finished GameObjects and Components to Entities to fully utilize DOTS will require quite some time.

As we can see, in the above situation the development time have to be extended because a critical component that we based on have to be completely changed to a more performant one. This will cause delay, increase of production cost and other undesired effects. The below graph illustrates why it is important to pre-identify possible critical components at an early stage in development from a financial point of view.

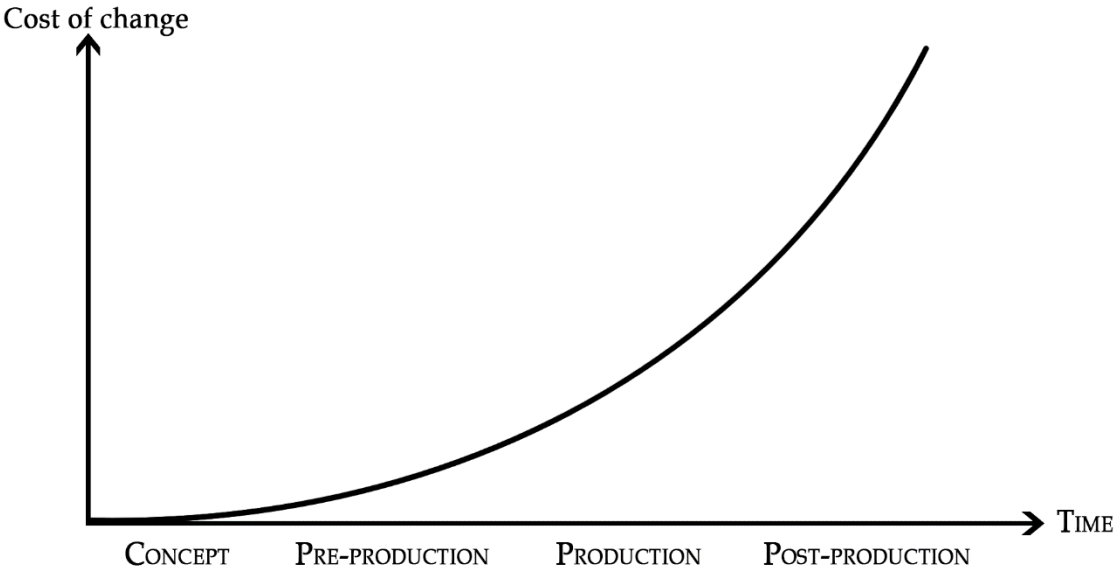

Figure 4: The cost of change in perspective of time

[11] **E**ntity **C**omponent **S**ystem, is a data-oriented design view opposite to Unitys's standard object-oriented one with GameObjects and Components

2.3.3 Production

Production is the longest phase in the development pipeline. This is the phase where we bring the project to life. Most of the well-known titles are in production phase for years. In this phase

- The level designers design the environment in a way that it perfectly fits into the gameplay.
- The programmers write numerous gameplay scripts, engine extensions and editor scripts to help their team iterate faster.
- Audio engineers record and create unique sounds for every detail in the game.
- Artists are creating characters, environmental props, and unique textures to fit our art style.
- Animators are putting life to these lifeless polygon monstrosities.
- And finally, the project lead trying is to coordinate the work of all these awesome and talented people.

And we did not mention voice actors, actors, script writers, composers and more. As we can see, this is the most exciting and hardest part to coordinate and iterate. It is also quite common that months worth of works have to be undone because it does not fit into the bigger picture. This is a quite frustrating phenomenon that can deeply demoralize our team, so preventing it with proper planning and communication is crucial.

When the team is finished with this phase a fully playable alpha or beta version is produced depending on the internal pipeline and testing in the large can happen. This is also a good phase for marketing because we will have plenty of nearly full done features to be shown.

From an optimization point of view, we will define a plenty of optimization goals in this phase. This is also the phase where we usually try to set up a more precise hardware requirement for PCs, and mobiles. We have a playable alpha or beta version so we can measure the game's performance on different hardware. We can also collect information from alpha/beta testers about the game's performance. On this collected information we will then define new optimization goals and later, usually in the post-production phase, we will then define the exact minimum/recommended hardware requirements.

2.3.4 Post-production

In this four-phase development pipeline post-production contains both just before launch and after launch post-production.

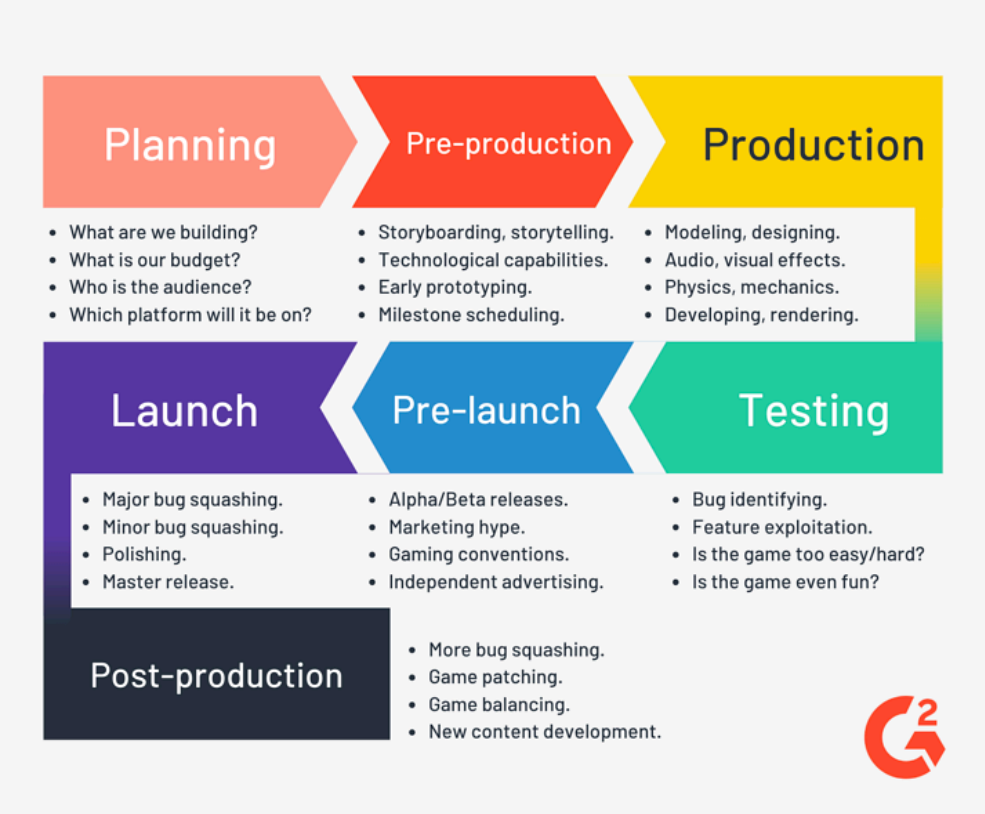

Figure 5: A seven-phase game development pipeline where post-production is separated into three phases. Pre-launch, launch, post-production. [4]

Here any type of big change is discouraged, highly not recommended and usually a sign of a disaster. The first half of this phase is about applying the final touches to our game before launch. At this stage

- The marketing team is working hard to market our game
- Some of the team members are adding small requested features (usually QoL[12]) that came up while beta testing
- Some of them are optimizing different parts of the game for smoother gameplay experiences
- But most of the development time is mainly spent on bug fixing

[12] **Q**uality **o**f **L**ife improvements usually make the gameplay more ergonomic and user-friendly

After the game is launched post-production should still go on because

- The game might need emergency patches (fails to launch, game breaking bugs etc.).
- The game needs patches that apply balance and general bug fixing.
- The game might even get new contents.

In some cases, post-production is a "never-ending" process. E.g. in an MMORPG[13] after launch the core game will be in post-production till the servers finally get shutdown. An MMORPG needs continuous support for accounts, moderation, in-game bugs and more. The developers also work actively to make the servers feel like a living world by adding seasonal events, small content patches and more.

From an optimization point of view, we should have only minor optimization goals. However, this is the phase where we should define precisely the minimum/recommended hardware requirements that will be printed on the "boxes".

[13] **M**assively **M**ultiplayer **O**nline **R**ole-**P**laying **G**ame is a genre that combines RPG elements with a vast number of concurrently online players

3 Unity

3.1 About Unity

Unity is a popular real-time development platform developed and maintained by Unity Technologies. It is mainly used for game development but in the past three years the industrial area showed serious interests towards it as well. The real success of Unity is in its portability (**Figure 6**), ease of use and the vast amount of tutorials and education materials made by its community. In 2019 53% of the top 1,000 grossing mobile games and in overall 50% of all games were powered by Unity.

Figure 6: The supported platforms of Unity 2020. Even though the next-generation consoles are not yet released to the public Unity is already supporting them. [5]

Figure 7: Some of the well-known titles made with Unity

It is a common practice that a game engine specializes for a genre(s). E.g. HeroEngine specializes for MMORPGs and RPGs, CryEngine and Frostbite mainly specializes for first-person shooters and Cocos2d-x is exclusively developed for making 2D (and nowadays 3D) mobile games. However, Unity is an exception to this.

This can be clearly seen in **Figure 7** where Cuphead is a 2D platformer, Fall Guys is a physics-based 3D battle royale platformer, Escape From Tarkov is a multiplayer FPS[14] and Hollow Knight is a 2D action-adventure game. All these games require different technology stack yet Unity can offer them all. Some of the main features of the engine includes:

- Animation and cinematic tools
- Asset store
- VR[15], MR[16] and AR[17] support
- Editor
  - Customizable Editor UI
  - Package Manager – NPM and NuGet like package management for editor extensions
  - Native import of well-known software file formats – Photoshop, Maya, 3dsMax etc.
- Programming Tools
  - IDE support – Visual Studio, Visual Studio Code, Rider
  - Unity Test Framework – A test framework for Unity
  - Unity Profiler – A profiler tool for performance analyzation
  - Detailed API Documentation
- Rendering
  - Various Render Pipelines – Different render pipelines for different requirements
  - Visual Shader and VFX editor
  - Post-processing stack
- Multiplayer and Networking
- Navigation and Pathfinding
- Physics
- User Interface
  - Various UI solutions for both in-editor and in-game UI scripting
- 2D
  - Sprite editing
  - Layer sorting
- World Building
  - Terrain System – A highly efficient heightmap based terrain system with various tools
  - Polybrush – Allows to blend textures and sculpt meshes directly in the editor
  - ProBuilder – A 3D modeling and level design tool for fast prototyping

---

[14] **F**irst-**p**erson **s**hooter is a game genre centered on weapon in a first-person perspective, so the player can experience the action throught the eyes of the protagonist

[15] **V**irtual **R**eality is a type of application which simulates an environment around the user

[16] **M**ixed **R**eality is a type of application which combines the simulated environment with the user's own environment allowing interactions between them

[17] **A**ugmented **R**eality is a type of application which projects part of the digital world over the real world

The aforementioned list of main features supposed to illustrate how versatile the Unity engine is. However, it is important to point out that these features are collected from the LTS version of Unity 2019. Unity releases four major builds every year. Three tech releases and one Long-term support (LTS). (This is going to change with Unity 2020 and up, where there will be three major releases, two tech and one LTS). Tech releases are supposed to show off new completed features to the users while LTS releases are maintained for a longer period of time than tech releases, however they contain no new features just usability and stability improvements. Between the tech releases there might be breaking changes[18], deprecated APIs or completely new features thus, every serious project should use the LTS versions of Unity. In this thesis we will use the LTS version of Unity 2019.

Unity offers four type of plans for developers namely Personal, Plus, Pro and Enterprise. Personal and Plus have restrictions on annual revenue meaning that only those are eligible for using them that meet those criteria. Personal and Plus also have limitations on official support services by Unity like collaboration tools, build servers and more[19]. Pro have little to no restrictions and Enterprise is basically an upgraded Pro license for bigger companies with dedicated contact to Unity. It is important to note that even the free Personal license has no restrictions on the engine itself meaning that even the smallest team can achieve anything without spending a single penny on Unity. The only drawback of the Personal license is that the splash screen[20] have to contain the Unity logo. In this thesis we will use the Pro version of Unity.

[18] A type of change in a software system that potentially causes other parts of the system to fail.

[19] https://store.unity.com/compare-plans

[20] The screen that is shown when the Unity app loads

3.2 Overview of the workflow

Unity's main workflow revolves around GameObjects and Components. GameObjects are the actors or we can think about them as container of Components while the Components define how a GameObject behaves in the scene. This workflow is extremely convenient and easy to understand, the developer just creates a GameObject, adds some Components to it then these will define how the GameObject behaves in the scene. E.g. adding a Rigidbody component to a GameObject will allow it to take part in physics calculations and if the developer clicks on play, the GameObject will instantly start to fall according to the laws of gravity. This is the main workflow of Unity. However, since Unity 2018 a new workflow was introduced called DOTS. DOTS is pushing Unity towards a data-oriented design view instead of the standard object-oriented one. Unity suggest that we should organize our code around data and the way how it is stored in memory. This results in an extremely performant code that can fully harvest the capabilities of modern hardware, via caching data efficiently. E.g. when a developer wants to iterate over all the zombies in the game in order to move them the CPU will load a bunch of ZombieController scripts into its cache. The problem is that ZombieController contains not only the position of the zombie but also other various properties like health, stamina, behavior type and attack type. Obviously for movement translations the only relevant property is position, yet the CPU loads all the irrelevant properties as well thus wasting our precious and extremely fast L1, L2, L3 caches.

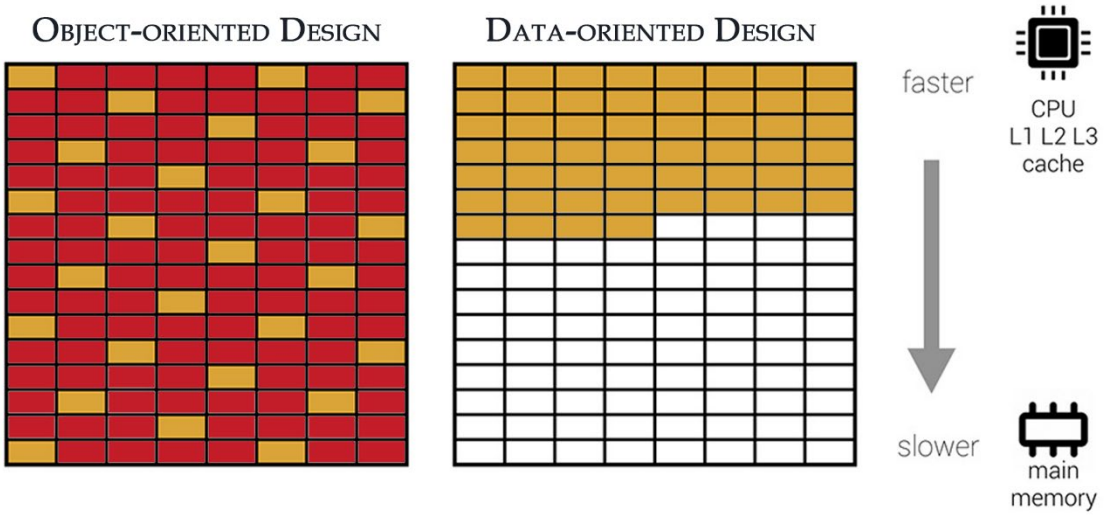

Figure 8: The difference between Object-oriented and Data-Oriented design [6]

As we can see in **Figure 8** object-oriented design uses an unoptimized data layout which brings irrelevant data (red) into the CPU cache thus it has to use the slower main memory. On the other hand, data-oriented design uses an optimized data layout that only brings the relevant data into the cache resulting in a half-empty CPU cache which is ready to cache more “zombies”. What the developers at Unity do since 2018 is that they are rewriting the core of the engine to harvest the full power of data-oriented programming. However, in spite of the promising results and performance boost, by default, we are not going to use this approach in this thesis because most of the DOTS features available to the developers are still in early preview thus expected to change frequently and significantly while containing quite many bugs according to my experiences.

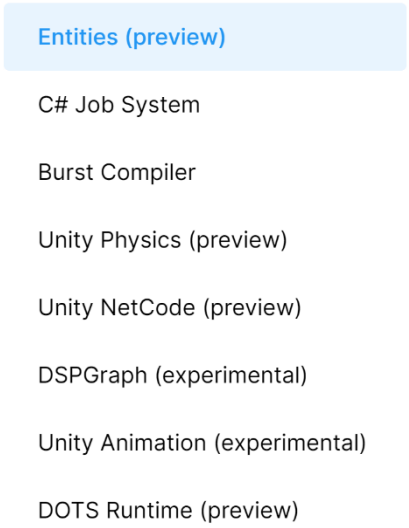

Figure 9: The packages of DOTS. [7]

4 Scripting in Unity

Scripting is an essential part of game development. Its main purpose is to decouple the game engine layer from the game logic layer, therefore allowing the management of both layers independently. In order to fully understand how scripting works in Unity we will look into the following topics:

- C#, the scripting language of Unity
- .NET in Unity
- Scripting backends.

Unity is a C/C++ engine, which means that most of the inner functionalities are written in these two languages. However, Unity's scripting language is C#, which means that the game logic will be probably written entirely in C#. This tends to confuse newcomers since interactions with the Unity API happen via C# objects (wrapper classes around the native C/C++ libraries), they think that the engine itself is written in C# as well. So, it is essential to know that this is not true and this knowledge is especially crucial if our task is to optimize parts of our scripting logic. Unity's source code is also proprietary which means that any problem that is on Unity's side have to be solved by them thus the developers are sometimes depending on the Unity team. Although, in 2018 Unity Technologies decided to release the source code[21] of their .NET assemblies under a reference-only license[22] which helped to find many bugs in these assemblies via the help of the community, we still don't have much insight into the inner mechanisms of the engine (C/C++ part). Moreover, Unity does not accept pull requests or issues on the GitHub repository which means that every bug has to be reported via Unity's official bug tracker called the Unity Bug Reporter.

But why Unity used C# and not C++ which would require no wrapper classes around the engine libraries thus avoiding language interoperability or why did they not choose Java, an older and library rich language that is cross-platform by default or another C# like language? In order to answer these questions first let's see what C# is.

---

[21] https://github.com/Unity-Technologies/UnityCsReference/tree/2019.4

[22] Reference-only license means for the sole purpose of inspecting functionality to understand or improve performance of your games, applications, software, or other content developed with the Unity Engine.[8]

4.1 C#, the scripting language of Unity

C# is a modern general-purpose, multi-paradigm[23] programming language. It has its roots in the C family of languages and usually shares similar syntaxes with them. C# is mainly an object-oriented and component-oriented programming language. It aids these paradigms on a language level making it a natural language in which to create and use components. Since its origin, C# has added features to support new workloads and emerging software design practices like functional, generic or dynamic programming, making it a truly flexible programming language. C# was developed as one of the programming languages for the Common Language Infrastructure (CLI) which is a standardized (ISO[24]/ECMA[25]) open specification developed and maintained by Microsoft that allows the use of different languages on different platforms without rewriting the code for that specific platform making CLI a platform agnostic solution. This thesis is too short to fully introduce C# and all of its features so we will concentrate on the areas that make it a perfect choice for game development.

In the introduction of the parent section we asked a simple yet important question

"Why Unity uses C# instead of C++?"

A short answer with the primary reason to this question would be "because of speed". Yes, C# is faster than C++, usually by around 3-12 months on larger projects. Jokes aside, C# is an extremely productive language and in game development iteration time is a crucial part of the development process. Many of the AAA games have an extremely long development time. E.g. the new "The Elder Scrolls" sequel is expected to be in development for six years. This means that for six years the studio will make no profit from the sequel only burn money on it. The cost of an engineer is not cheap, if we have the ability to boost their productivity just by changing the language they use that would be quite beneficial from a financial point of view. Remember that every game is a business product. The goal is to make an enjoyable and profitable game from as low cost as possible and generally C# will help us more to achieve this than C++. Let's see why.

---

[23] Programming paradigms are a way to classify programming languages based on their features. E.g. C# is an imperative, declarative, functional, object-oriented and component-oriented language.

[24] **I**nternational **O**rganization for **S**tandardization

[25] **E**uropean **C**omputer **M**anufacturers **A**ssociation is a standards organization for IT related fields.

One of the main reasons that programming in C# is generally more productive than in C++ is that C# is less error prone which means less debugging and bug fixing thus the programmers can focus on features rather than on bugs. This reliability comes from different features of the language and its runtime.

Firstly, C# is a managed language in contrast to C/C++. Managed languages are languages whose execution are managed by a runtime. In case of C# the runtime is called the Common Language Runtime (CLR). The CLR provides various services that makes the development process more productive. One of the most important services of the CLR is the automatic memory management. The CLR provides a Garbage Collector (GC) which takes care of releasing objects that are no longer used by the program. Garbage collectors nearly fully eliminate the need of manual memory management thus nullifying one of the most common sources of bugs called memory leaks. E.g. according to Microsoft and Google 70% of all security bugs are memory safety issues[13].

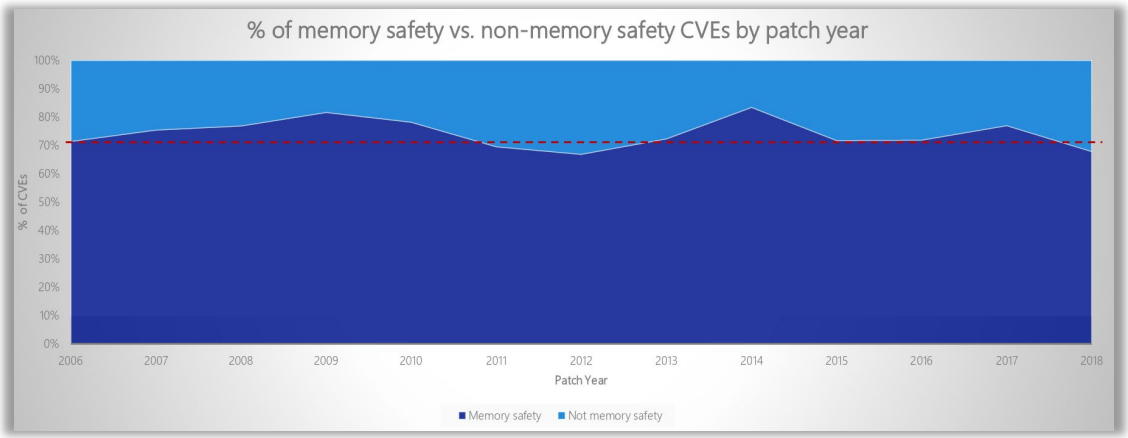

Figure 10: Microsoft's statistics about memory safety. According to Google at least 30% of the memory safety issues are "use-after-free" memory leaks [13]

Although, **Figure 10** shows only the security related issues, we can safely assume that memory related bugs are quite common around other types of bugs as well. C# also has other memory related security features like bound checking and enforcing variable initialization. In C++ it's the programmer's responsibility to assign a value to a variable before the first use. From a performance perspective this is good, (micro optimization wise) however it can cause undesired behaviors if the assignment is forgotten before the first usage. C# compilers enforces variable initialization through static checking preventing yet another common source of memory related bugs.

Another critical part where C# can outshine C++ is compilation time. Gameplay and general game scripting needs fast feedbacks about changes but C++ compilation takes time. One of the main problems when compiling a C++ project is the handling of header files. Every single compilation unit requires several header files to be loaded, parsed and compiled. This can produce a huge amount of work in case of big projects. C# on the other hand does not have header files. In addition to this C#, code compiles not to machine code but to an intermediate language called the Common Intermediate Language (CIL) (formerly known as MSIL[26]). These compiled codes will be then stored in assemblies (files with a .dll or .exe file extension). When the application is started the CLR loads these assemblies and uses a just-in-time (JIT) compiler to translate the CIL into machine code that can be directly executed on the underlying architecture.

We can illustrate the boosted productivity via compilation time by an extremely simple example. Let's say our team consists 10 gameplay developers who test their code changes via the editor play mode 10 times a day (a generous underestimation). They are working on a DLC to a huge scale AAA game with an enormous C++ code base. The expected production time is estimated to be a year. The compilation of the scripts on average takes around 60 seconds regardless of where and what changes were made. Again, this is a quite generous underestimation for compilation time for large C++ projects but let's assume that the team uses the best practices to reduce compilation times for their project. (Entering play mode also require some time depending on the project but since it can highly vary from project to project, we exclude it from the calculation.)

$$Number\ of\ devs * Daily\ compilations * Working\ days\ a\ year * Average\ compilation\ time$$
$$10 * 10 * 230 * 60 = 1\,380\,000\ seconds$$

Our gameplay developers waited for 1 380 000 seconds just for compilation. 1 380 000 seconds is roughly 384 working hours. There are 8 hours in a working day. (Let's ignore the fact that many AAA studios basically requires the employee to do overtime.) This means that each developer individually spent nearly 5 working days waiting for the project just to compile. C# projects on the other hand usually compile at least twice as fast as C++ projects, but 3-4x+ faster compilation times are not rare either. This would mean that if the game were scripted using C# each developer would only wait 2.5, 1.5 or

[26] **M**icrosoft **I**ntermediate **L**anguage

less than just 1 day for compilation overall. This might sound great but still not the 3-12 months desired productivity boost. But we must not forget that this is just compilation time in a one-year span only (with extremely generous underestimations).

In the introduction of the parent section there was another important question that is still unanswered.

> "Why did they not choose Java an older and library rich language that is cross-platform by default or another C# like language?"

The answer to this question is not a trivial one. Several factors could have influenced Unity's initial decision like target platform support, licensing of the language etc. In this thesis we will concentrate on the language, framework, and libraries that could have influenced Unity Technologies' decision.

C#, as mentioned above, is a managed language which also uses a GC for automatic memory management. This comes with many safety and convenience features, however there is one big problem, garbage collection takes time. E.g. a competitive first-person shooter game made with a GC backed language runs at 144 FPS[27] then suddenly the FPS drops to 72 for a split second. This is a common problem and usually indicates that the game allocates too many objects on the heap[28]. Everything that gets allocated on the managed heap the GC will take care of. This means that if we allocate too many objects on the managed heap the GC will have to run frequently thus dropping our frame rate. Luckily C# can help us to avoid unnecessary heap allocations in various ways. C#, unlike Java, distinguishes two main types from the perspective of the memory. Reference types and value types. While reference types are always allocated on the heap[29] value types are allocated on the stack[30] with some exceptions. The two most important exceptions when a value type is actually allocated on the managed heap are

- The value type is a field of a class.
- The value type is boxed[31].

---

[27] **F**rames **p**er **s**econd. In Unity fps can be calculated by using the Update message.

[28] Heap is the portion of the memory where the dynamically allocated memory resides.

[29] Java compilers might allocate reference types on the stack if they can prove that the reference won't "escape". The technique is called escape analysis.

[30] Stack is an array of memory in a LIFO (Last In First Out) structure.

[31] Boxing is when the CLR wraps a value type inside a `System.Object` instance and stores it on the heap.

But why is it good that value types are stored on the stack rather than on the managed heap? Because of efficiency. Deallocating and allocating on the stack are extremely cheap compared to deallocations and allocations on the managed heap. Moreover, types allocated on the stack are not subject to the GC. Since value types are quite commonly allocated on the stack, by using them cleverly, we can take a lot of pressure off from the GC. The problem is that many languages only support this behavior on primitive types[32]. Luckily, C# supports structs that happen to be value types that support encapsulation of data and related functionalities therefore allowing developers to define class like behaviors. There are some golden rules on when and how to define structs. E.g. it is advised to define them as immutable structs because of their value type semantics. According to Eric Lippert former designer of the C# language

"Mutable value types are evil. Try to always make value types immutable."[14]

Since C# 7.2 the language have an extended syntax in order to give opportunity to enforce immutability and allocations to stack only.

Now let's see a common usage of these structs in Unity. In Unity every GameObject has a Transform component. It is impossible to create a GameObject without it, or delete it from an existing one. One of the most common scenarios in a Unity app is movement translation.

```
public void MoveUpByOneUnit()
{
    Vector3 pos = transform.position;
    transform.position = new Vector3(pos.x, pos.y + 1, pos.z);
}
```

The above method moves up the current GameObject by one unit. In Unity `Vector3` is a struct. Every time `transform.position` is called Unity queries the current position of the GameObject from the C/C++ part of the engine. If `Vector3` was a class it would mean that every time we query the position, a `Vector3` object would be allocated on the managed heap. In case of an Update loop hundreds of instances of this class would be instantiated in every second, slowly pressurizing our GC. However, since `Vector3` is a struct it gets allocated on the stack and when we leave the scope of the method it gets

[32] int, long, float, double, bool, char, enum etc.

deallocated cheaply without the need of the GC. This type of behavior is especially crucial in game development and generally optimization-wise.

In the above movement translation example `transform.position` is a property. However, there is one special thing about this property. If we look into the `Transform` component's source code, we will see an `extern` keyword in the property declaration.

```
// Position of the transform relative to the parent transform.
public extern Vector3 position { get; set; }
```

The extern keyword indicates that the method is implemented externally. In case of Unity this means that this method is implemented internally in the C/C++ code base. This leads us to another reason why Unity settled with C# rather than with another managed language like Java. C# and the .NET ecosystem have a strong native interoperability. C# was designed for cross-language support. It needs no third-party solutions for interoperability the CLR and System libraries provide all the necessary resources for cross-language support. C# supports interoperability with language level syntaxes. The framework provides marshalling[33] services, various interoperability solutions like Platform invoke or shortly P/Invoke and convenient attribute syntaxes for library definitions. These services are extremely useful in game development. They make it possible to use C/C++ libraries from a managed code therefore allowing us to move performance critical codes outside of the managed environment or just simply allowing us to use popular libraries/services that are written in another native language. E.g. Epic Games, the company behind Unreal Engine made a so-called Epic Online Services. These services provide match making, server hosting, app store and other various services. However, these are all written in C++. Luckily this is no problem in case of C# and .NET. Epic Games just wrote a wrapper library around the public interfaces using one of the interoperability solutions just like Unity wrote wrapper classes around the engine's public interfaces. This is a common technique and many of the third-party libraries for Unity are actually C++ libraries with a C# wrapper layer.

[33] Marshalling is the process of transforming an object's memory layout into another suitable memory layout for transmission. In C# it is usually reffered to converting managed types to unmanaged ones.

Although interop calls are extremely useful and they are quite crucial in game development they do have a significant overhead compared to simple method calls. E.g. P/Invoke has an overhead of 10-30 x86 instructions per call in contrast to a simple call instruction [15]. In addition to this, marshalling can also add additional overheads to this depending on the data that needs to be marshaled. Luckily in the above example `Vector3` only contains blittable[34] types therefore `Vector3` requires only minor marshalling. A sequential struct layout attribute is needed and our `Vector3` struct can be marshaled as a C structure.

4.2 .NET in Unity

.NET is a developer platform made up of different kinds of tools, programming languages (C#, F#, Visual Basic) and a vast amount of libraries for building any kind of application. There are various implementations of .NET and every implementation of it is actually the implementation of the Common Language Infrastructure (CLI).

- **.NET Framework**
  The first implementation of .NET. Although .NET Framework is platform independent in theory it is only implemented on Windows thus it only supports Windows operating systems.
- **.NET Core**
  .Net Core is the cross-platform implementation of the CLI and the successor of the .NET Framework. Unlike .NET Framework, it is open-source and maintained and developed by Microsoft via the .NET Foundation. (Unity is part of the .NET Foundation)
- **Mono**
  Mono is an open-source cross-platform implementation of .NET Framework. Mono can be run on platforms like Android, iOS, tvOS, Linux, Windows, macOS, PlayStation 3, PlayStation 4, Xbox 360, XboxOne, Xbox Series X, Wii and many more. Mono does not support all of .NET Framework's API but the core API is fully supported.

[34] Blittable types are data types which memory representation is the same in both managed and unmanaged memory therefore requiring no marshalling. Some well-known blittable types are byte, int, float, double.

The above three are complete frameworks. Although if a developer goes into Unity's Project settings under the Player subsection there will be two selectable API Compatibility Levels, .NET 4.x and .NET Standard 2.0 and the Scripting Backend which is set to Mono as can be seen on **Figure 11**.

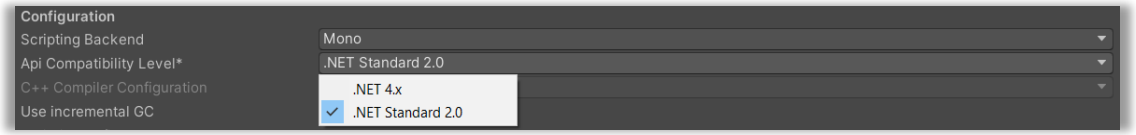

Figure 11: The Player settings in Unity's Project Settings.

This can be confusing for inexperienced .NET developers since .NET Framework versions are sometimes just called e.g. .NET 4.7 etc. The keyword(s) here is the API Compatibility. While Mono is the framework the API Compatibility Level defines the available libraries. .NET Standard 2.0 is not a CLI implementation and a complete framework, it is just a specification of .NET APIs that are available on all .NET implementations. Perhaps a picture tells more than a thousand words. (**Figure 12**)

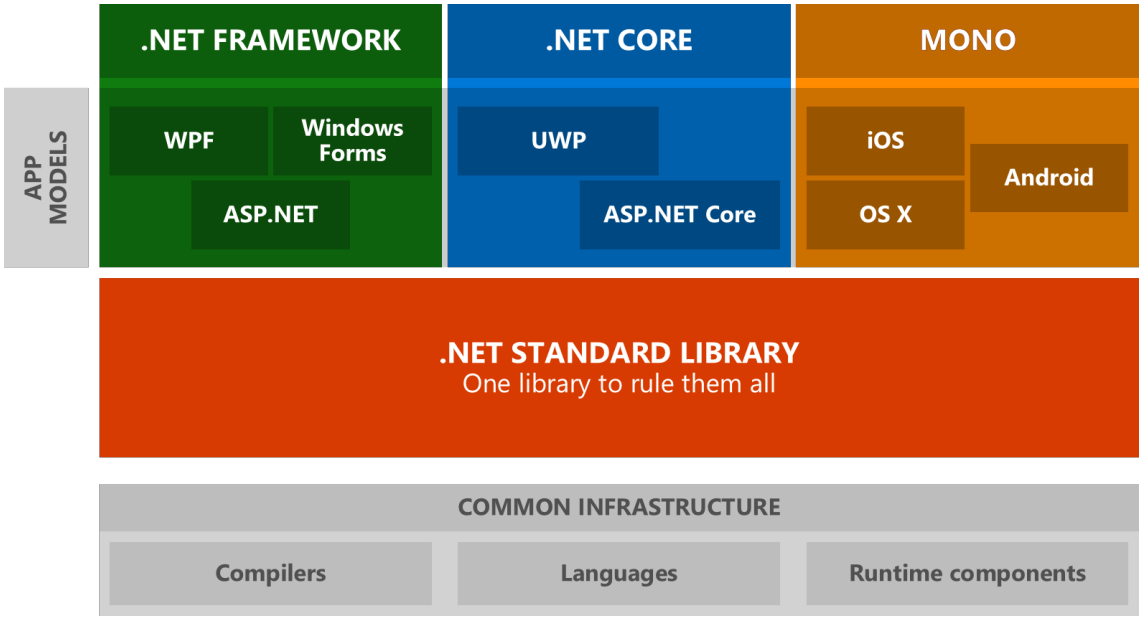

Figure 12: The purpose of .NET Standard.

The general rule of thumb in Unity is to use .NET Standard 2.0 and only switch to .NET 4.x whenever the project needs specific API(s) defined in it. The reason for this is because .NET Standard 2.0 have a smaller library which compiles faster and produces a smaller build size that is quite useful for mobile applications.

In conclusion Unity uses Mono as it is the framework that provides them cross-platform support on a high range of devices and .NET Core does not yet supports that many platforms as Mono. However, in November 2020 .NET Core will be merged with .NET Framework and will be developed as an open-source cross-platform framework with the name .NET. (First version will be .NET 5.) In 2021 .NET will be merged with many of the functionalities of Mono (.NET 6) therefore Unity sooner or later should switch to .NET (Core). .NET Core is such a powerful open-source project that already some of the principal engineers of Unity tried to port the scripting backend to .NET Core. [17]

4.3 Scripting backends

Scripting backend is a Unity term for the framework that will compile and execute our code. In Unity 2019 there are 2 scripting backends. We are already familiar with the one called Mono and the other one is named IL2CPP (Intermediate Language to C++). IL2CPP is a Unity developed scripting backend. It contains an AOT[35] compiler and a runtime library to support the virtual machine. Both scripting backends have some advantages and disadvantages.

4.3.1 The Mono scripting backend

The Mono scripting backend is the backend of the editor. There is no way to use IL2CPP when testing a game via the editor. This might sound inconvenient when the final builds will be all compiled via IL2CPP but as we talked about it earlier Mono provides a seriously faster iteration time than IL2CPP since compilations are much faster with it (involves no C++ compilation and because it only gets compiled to an IL). Mono as a JIT based scripting backend also supports dynamic C#/IL code generation at runtime. However, that are many cases where Mono is not a suitable scripting backend. There are some platforms that do not allow runtime code generation therefore any JIT based managed code will be prevented to execute on the target platform. In this case an AOT compiler is required. A good example for this are iOS devices.

[35] **A**head-**o**f-**t**ime compilers compile the code into static libraries that can be then directly executed on the target architecture without the need of dynamic compilation like in the case of a JIT compiler.

4.3.2 The IL2CPP scripting backend

IL2CPP is Unity's own scripting backend first introduced in Unity 5. Since then it has undergone serious changes and now it is one of the main scripting backends of Unity. Some of Unity's supported platforms can only be shipped with IL2CPP. IL2CPP's biggest advantage is in its configurability, framework independent GC and in its performance. Although IL2CPP is generally faster than Mono there are some cases and platforms where it can be considerably slower.

IL2CPP first compiles IL code from the scripts assemblies into C++ then compiles these to native code which can be directly executed on the target architecture, this can be seen on **Figure 13**. Libil2cpp is the name of the runtime library (e.g. this contains the GC) and IL2CPP.exe is the compiler. The compiler is an AOT compiler.

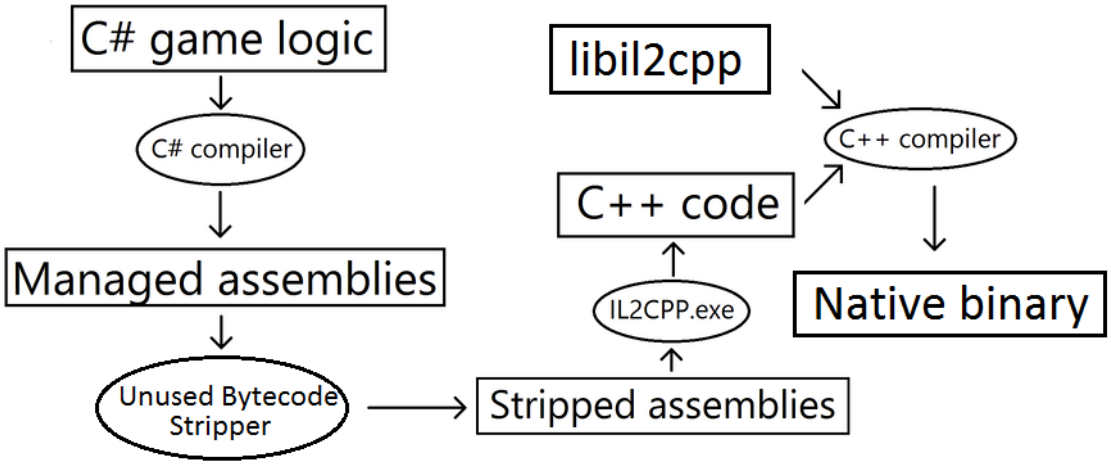

Figure 13: The compilation process of IL2CPP [18]

It is important to point out that the standard libraries of .NET are not transpiled to C++, mscorlib.dll, System.dll, etc. are the exact same code used for the Mono scripting backend. The converted C++ code currently uses the Boehm-Demers-Weiser garbage collector.

IL2CPP has some restrictions on reflection[36], C# exception filters and on some other minor features. Unity Documentation has a detailed page dedicated to these scripting restrictions.[37]

[36] Reflection is a process to obtain information about loaded assemblies and the types defined within them

[37] https://docs.unity3d.com/Manual/ScriptingRestrictions.html

5 Performance analysis in Unity

Now that we are familiar with common performance issues and considerations in game development and we know how to approach and assign optimization goals to them and that we have also become familiar with the scripting ecosystem in Unity it's time to talk about how to handle these in practice inside Unity.

In this section we will discover what types of tools are available for performance analysis in Unity, how to use these tools and identify the bottlenecks and hot spots[38] with the help of them. Then we will discover what types of tools are available in Unity to write performance tests and benchmarks in order to maintain performance requirements or just to select the most optimal solution to a specific problem.

5.1 Profiling

Profiling in computer science is a form of dynamic program analysis where we measure different parts of our application. Whenever a performance degradation is observed a performance profiling should take place. It is a common mistake amongst rookie programmers that only assumptions are made thus it can happen that a subsystem gets optimized that is not even involved in the root cause. It is needless to say that this approach is highly inefficient. There are several tools available in Unity to help us identify performance related issues like the

- Profiler
- Memory Profiler
- Frame Debugger
- Physics debugger.

Moreover, there are other tools available that can be used with Unity like Visual Studio's profiler or for platform dependent optimizations the corresponding tools. E.g. for Android the Snapdragon Profiler.

In this thesis we will concentrate on Unity's Profiler tool and on the official packages made for performance analysis like the Performance Testing Extension.

[38] A hot spot in computer science is most usually defined as a region of a computer program where a high proportion of executed instructions occur or where most time is spent during the program's execution.

5.1.1 The Unity Profiler

The Unity Profiler is the most commonly used profiling tool for Unity applications. It can be connected to devices over the network or to devices connected to the machine physically allowing the developers to analyze the application on their release platforms. The profiler can be also connected to the editor allowing analysis of in-development features. The profiler gathers and displays data on the performance of the application such as CPU, GPU, memory, renderer, audio, physics, network etc. The profiler displays the gathered data in both text and charted view as well.

The Profiler should be the first tool for performance related issue when developing with Unity. It has four main windows as it can be seen on **Figure 14.**

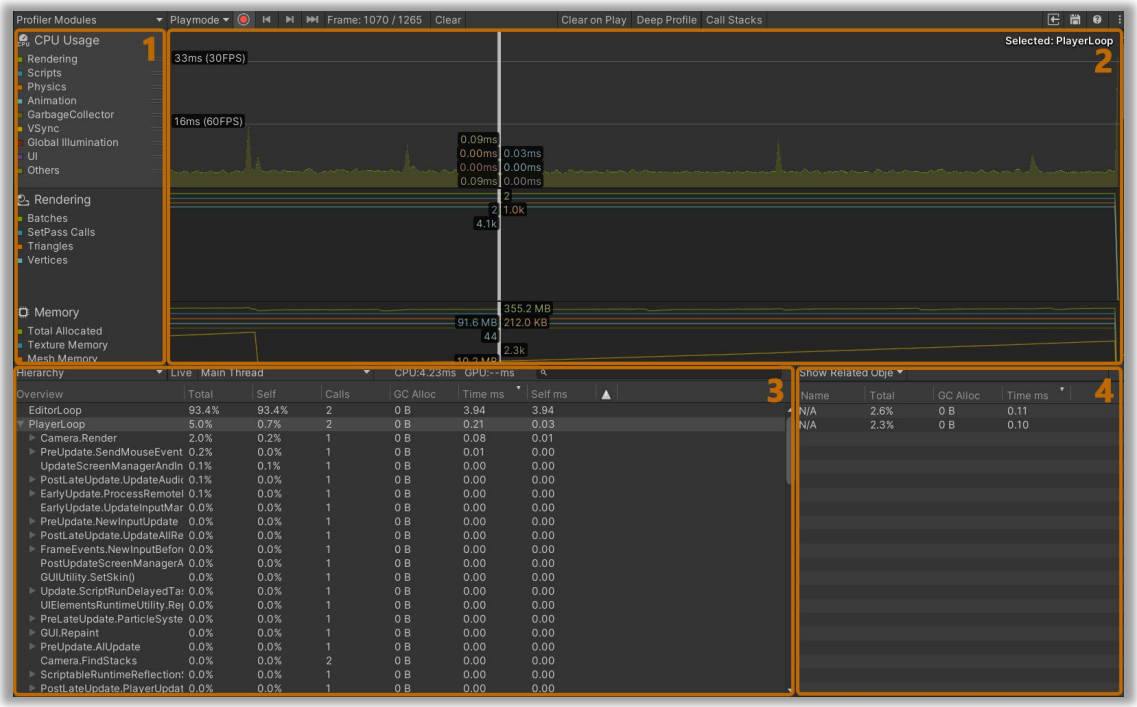

Figure 14: The Profiler window

1st is the Module window. We can select here which module we would like to inspect. (CPU, memory, rendering etc.) **2nd** is the Profiler area window where we can see the measurements of the modules frame by frame in a charted view. **3rd** is the Overview window where we can see a detailed breakdown of each module. Currently the CPU is inspected. **4th** is the Object display window where we can get an information about the objects that are related to the inspected element. As we can see currently the player loop is selected therefore there are no user related objects displayed in this view.

In this thesis we will mostly use the CPU module since scripts are most of the time bottlenecked by the CPU. The CPU usage window is measured frame by frame. Since game engines are working in so called game loops this is a perfect fit for performance analysis. We can inspect where and what the CPU was working on in each frame. The Overview window contains all the related methods and functions that were executed in the currently selected frame. On **Figure 14** the CPU's work was mainly in the Editor loop[39]. The reason for this is that the picture is from an empty project containing only a camera and a directional light. But we can see that the CPU spent some time rendering, sending mouse events however, most of the work falls into the Others category[40].

It is important to note that profiling through the editor only gives approximate results especially if the target device is not a PC or Mac but a console or a mobile device. So, profiling through the editor is mostly useful when there is a non-platform dependent issue that requires no exact measurements, an in-development feature is analyzed or the target platform is the platform where we run our editor and the editor overhead is not expected to add a significant distortion to the results. E.g. 10 000 objects could significantly affect both the editor and the player since the editor would need hierarchy updates, gizmo draws, scene view draws etc.

5.2 Performance testing

It is a common mistake that after a performance degradation observation the component that was causing the degradation gets optimized and then later in the development process the problem emerges again. This can be easily prevented by performance testing. After solving a performance issue there are some cases where we should verify the optimized component via performance tests in order to avoid a common scenario in which a change to the component falls its performance back below the accepted minimum. E.g. if a level loading issue emerges which makes level loading way too long and a performance/optimization goal was already defined for the problem we should ensure that the goal will be kept through the development process. If the goal is to keep loading times under 2 seconds on the target platform then this should be verified

[39] Editor loop in the profiler is the editor related overheads like drawing the hierarchy, inspector view etc.

[40] The Other category in the CPU module are mostly engine related updates, like coroutine manager updates, render pipeline handlers etc.

through performance tests. Performance testing is also useful to provide information about the capabilities of our game's performance, so later we can fine tune application settings based on the results or simply compare previous results in order to define what changes affected our performance in what way.

Generally, performance testing does not mean that the results should be asserted. Performance testing has many branches like load testing[41], stress testing[42] and configuration testing[43] and none of this includes assertions generally. Their results are monitored and recorded but mostly analyzed manually. In this thesis we will however write asserted performance tests using the AAA[44] pattern. The reason for this is that most of the traditional performance test branches cannot be applied to game development. E.g. we don't do traditional stress testing since a low response game might work but it is unacceptable from the point of the user. On the other hand, writing asserted performance tests can automatically enforce previously set performance/optimization goals or just a reasonable performance requirement.

Sadly, performance testing is widely ignored in game development which can be seen even in AAA titles, however they are more common around indie titles. A good example to this is PlayerUnknown's Battlegrounds also known as PUBG. It was an indie game that gained massive popularity amongst players in a short amount of time. The huge player base had an ample set of system configurations and because the game already had various performance issues this just made things worse. Most of the performance issues were only fixed when the software giant Microsoft lent developers to PUBG in order to prepare the game for the Xbox release and did a comprehensive performance testing in the process.

In Unity we can use the Performance Testing Extension for Unity Test Runner for both benchmarking and various performance testing.

---

[41] The process of putting an expected load on the system and then measuring its response.

[42] The process of intense testing in order to determine the stability of the system. In game development mostly servers are stress tested and not the game itself.

[43] The process of testing the performance with various hardware and system configurations.

[44] **A**rrange **A**ct **A**ssert is a software industry standard testing pattern. In arrange we setup the test's prerequisites, in act we do the main action from the point of the test and finally in the assert section we check if the results are indeed the expected results. This pattern greatly helps readability.

5.2.1 The Performance Testing Extension API

The Unity Performance Testing Extension is an Editor package that provides an API and test case decorators to make it easier to take measurements and samples of Unity profiler markers outside of the Profiler or just to record custom measurements. The Performance Testing Extension is intended to be used with and extend the Unity Test Framework [45] (UTF). UTF's structure is similar to any Unit testing framework[46] therefore we are not going to detail its usage. UTF is built on top of Nunit[47].

The Performance Testing Extension (PTE) extends the UTF with one important `[Performance]` attribute. Every test that is marked with this attribute will be initialized as a performance test. The attribute is intended to be used with UTF's `[Test]` and `[UnityTest]` attributes. The former one is for non-yielding tests typically for those that execute under one frame while the latter one is for yielding tests that are executed across multiple frames. The PTE API provides various methods to take different kinds of measurements in our performance tests. It provides methods for warmups[48], iteration counts, GC usage, custom measurements and more.

In order to view the results of a performance test, the developer first needs to open the Unity Test Runner (Window > General > Test Runner). From here all the available tests can be run and inspected. However, there is no way to distinguish a performance test from the other types of tests here (unit, integration, component etc.) so a clear folder structure is advised for easier identification. When a performance test is executed the Test Runner will show a Test Summary. The summary consists of different type of information. Firstly, the summary includes every sample group's aggregated samples such as median, min, max, average, standard deviation, sample count, count of zero samples and sum of all samples. Secondly, the summary includes important information

---

[45] The Unity Test Framework enables developers to test their code in both Edit and Play Mode and also on target platforms such as Standalone, Android, iOS, etc.

[46] Unit testing is a software testing method where individual areas of the software are tested. https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2019

[47] A unit testing framework for .NET, initially ported from JUnit. NUnit is part of the .NET Foundation

[48] Warmup is mostly important in JIT based environments where the compiler can optimize the execution at runtime. Warmups can also help in non-JIT based environments to setup an optimal CPU cache etc.

about the application and its player settings, hardware information and editor information if available.

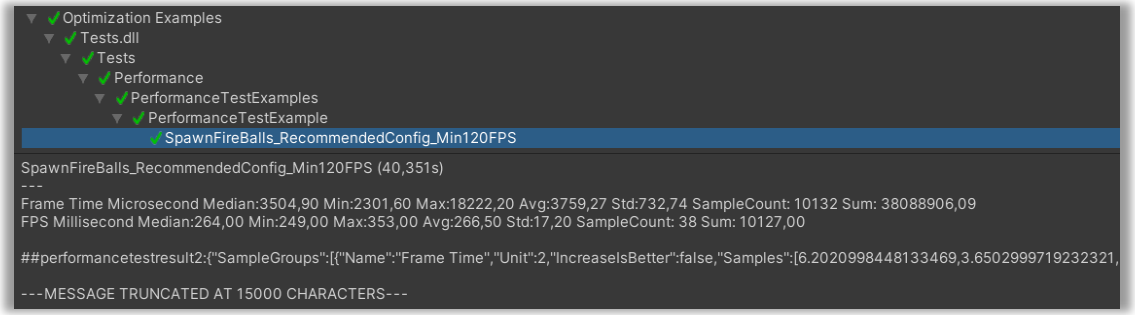

Figure 15: The Test Summary of a performance test inside the Unity Test Runner.

On **Figure 15** the green ticks indicate that the performance test named `SpawnFireBalls_ReccommendedConfig_Min120FPS` was successful. Moreover, we can see the Test Summary that contains the aforementioned information. However, there is one big problem with this summary window, the data is truncated in case of large data. Luckily, whenever a performance test is executed the results are saved to a platform specific location. In case of a Windows operating system these are saved into `AppData\LocalLow\<CompanyName>\<ProjectName>`. The results are saved into both `.xml`[49] and `.json`[50] which extremely popular in game development.

```
"Hardware": {
    "OperatingSystem": "Windows 10  (10.0.0) 64bit",
    "DeviceModel": "X470 AORUS GAMING 7 WIFI (Gigabyte Technology Co., Ltd.)",
    "DeviceName": "DESKTOP-RNN8BGG",
    "ProcessorType": "AMD Ryzen 7 3700X 8-Core Processor ",
    "ProcessorCount": 16,
    "GraphicsDeviceName": "NVIDIA GeForce GTX 760",
    "SystemMemorySizeMB": 16329
 }
```

Figure 16: The hardware info inside the performance test results `.json`. In this thesis we will use the above configuration for performance testing.

Although the results are saved into a human-readable file format our brain can't really process 10 000+ sampled values by just looking at the them so an external tool is advised that can visualize the data. Fortunately, there is an external tool called the Unity

[49] **E**xtensible **M**arkup **L**anguage is both a human and machine-readable data-interchange file format.

[50] **J**ava**S**cript **O**bject **N**otation, is both a human and machine-readable data-interchange file format.

Performance Benchmark Reporter. The Performance Benchmark Reporter produces a html report based on the persisted test results which can be opened in a browser for further examinations. Moreover, from Unity 2019 there is an editor integrated tool called Performance Test Reporter. This can also visualize the results of the executed performance tests allowing faster iterations since no external tool is required. The Performance Test Report tool can also export the data into the popular `.csv`[51] file format, allowing developers to easily import their results into tools like Microsoft Excel and Power BI[52]. In this thesis we will use the Performance Test Report tool for faster iterations.

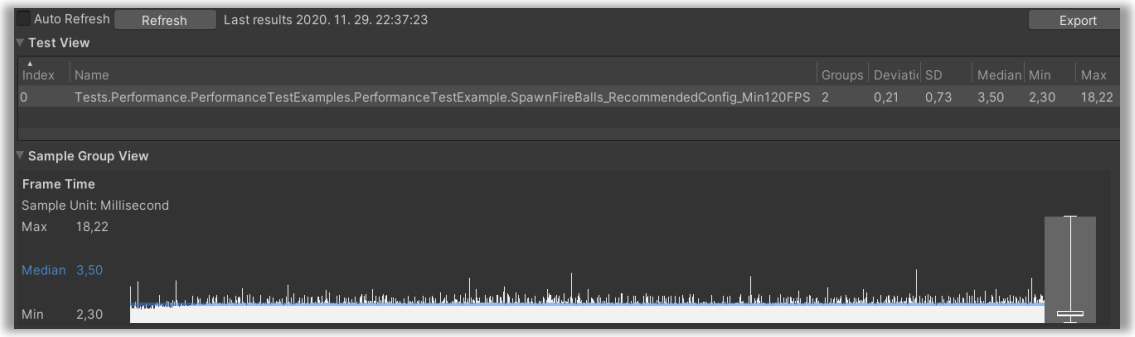

Figure 17: The Performance Test Report tool showing the visualized frame times.

One of the biggest advantage of this integrated performance tool family is that we can natively run these tests on the target platforms. Via `IprebuildSetup` interface we can implement a setup method that is automatically called before the performance test is executed by the Unity Test Runner. This allows the developers to flexibly build the player for performance tests using the same Unity project against different platforms, render threading modes, player graphics APIs, scripting implementations, and XR-enabled settings such as stereo rendering path and VR SDKs. Since this allows a common project for all target platforms a convenient way to run these tests is to use a command line. Using a command line allows the developers to pass arguments to the aforementioned setup method where these arguments can be parsed and a desired and customized player can be built for running the performance tests.

---

[51] **C**omma-**s**eparated **v**alues or **C**haracter-**s**eparated **v**alues is a common data exchange format which is supported by many applications

[52] Power BI is robust service for data visualization that can easily create reports and dashboards.

5.2.2 Performance test examples

One of the easiest ways to illustrate how useful performance testing can be and how powerful the PTE API is, is to use illustrate it with examples. The code and the project we use in this sub-section can be found on the GitHub repository[53] of this thesis.

5.2.2.1 Enforcing loading times example

It is a common performance and optimization goal that the game should load fast. This can be crucial in both AAA titles and in simple mobile games. E.g. with long loading times we can easily break the illusion of a RPG thus the joy of the player. The same applies to simple casual mobile games. If the user has to wait 5-10 seconds for a new level to load where levels are changed frequently and we even put ads after each level (which we should absolutely not) then our user will be dissatisfied quite fast and our game will belong to the group of "installed for 10 minutes". While reducing the adds is a marketing strategy, optimizing the loading times usually requires an engineer. In this example we will analyze the performance test written for enforcing a previously set optimization goal for loading times. The example is located in the GitHub repository under, `Optimization Examples\Assets\Scenes\Performance Test Examples`.

The game consists of three extremely simple scenes.

- Main Menu – a simple menu scene with 2 level selection buttons.
- Dragon Level – a level where the player can shoot fireballs on a dragon.
- Empty Level – an empty level that can be loaded rapidly.

All the relevant scripts are located under the Performance Test Examples' Script folder. The `DragonMover.cs` script moves the dragon automatically on the terrain while the `FireBallSpawner.cs` script shoots fire-balls towards the dragon from the sky. The `MainMenuController.cs` handles the UI actions in the Main Menu scene.

At the beginning of the development there was a problem about loading the dragon scene. There were so many initialization logics in Awake and Start messages that loading the scene took several seconds. Because of this an optimization goal was defined for level loading times stating that "Levels should load under 2 seconds on the recommended configuration". Luckily the acceptance criteria (the team follows an agile development

[53] https://github.com/Menyus777/Game-engine-specific-optimization-techniques-for-Unity

process) required performance tests, so an asserted performance test was written in order to enforce the optimization goal through the development process.

It is important to structure our code logically but since level loading is not just a simple method but a complex process it cannot be placed under MainMenuController's performance tests in spite of the fact that level loading is exclusively handled by this script. Moreover, performance tests should be as close as possible to the real-world use-cases (with some exceptions e.g. breakpoint test) thus it is advised to execute them as a user would execute the specific scenario. The loading times performance test are located under the folder structure `Assets\Tests\Performance\Performance Test Examples` inside the `LevelLoadingPerformanceTests.cs` script.

The Unity Test Framework is built on top of Nunit therefore the whole testing concept is the same as in Nunit. An instance of the test is created, setup methods are called and after all of the tests are finished the tear down methods are executed. The Unity Test Framework extended the available setup and tear down handlers so they are fit into the engine scripting mechanism. E.g. the code snippet on **Figure 18** shows a Setup method. We can clearly see this by the method name and by the attribute placed on the method. `UnitySetUp` is a special attribute that allows yielding[54] in the setup method therefore allowing coroutine[55] behavior in setup methods. This is especially crucial at scene loading since scenes are only fully loaded in the next frame rather than in the frame loading was originally initiated.

```
[UnitySetUp]
public IEnumerator BeforeEach()
{
    // Loading the relevant scene
    SceneManager.LoadScene("Performance Test Examples - Main Menu");

    // Wait a frame for scene load
    yield return null;
}
```

Figure 18: The setup method for the loading time performance tests

[54] In Unity coroutines are implemented via enumerators therefore the yielding logic in **Figure 18**.

[55] Coroutines are a type of methods which can pause execution, save state, then yield control back to Unity's game loop, so later in time the coroutine can continue execution where it "left off".

For loading time performance testing we use a UnityTest which is basically a test that will run in parallel to updates rather than just calling a method and stepping through it sequentially. Since UnityTests are yielding tests, rather than returning `void` they return an `IEnumerator` allowing on-demand yielding logic.

```
[UnityTest, Performance]
public IEnumerator LevelsLoadUnder2Seconds_RecommendedConfig(
    [ValueSource("_levels")](string levelButton, string sceneName) level)
{
   // Arrange
   // Creating a sample group for loading time
   var sampleGroup = new SampleGroup("Load Time", SampleUnit.Millisecond);
   var levelButton = GameObject.Find(level.levelButton).GetComponent<Button>();
   Stopwatch sw = new Stopwatch();
   // Stopping the stopwatch when the scene gets loaded
   SceneManager.sceneLoaded += (scene, loadMode) => sw.Stop();
   // Small idling aka warmup before measurement starts
   yield return new WaitForSecondsRealtime(2.0f);

   // Act
   sw.Start();
   // Clicking on the appropriate scene button
   levelButton.onClick.Invoke();

   // Waiting till the scene loads (sw is stopped when the scene is loaded)
   yield return new WaitUntil(() => !sw.IsRunning);
   Measure.Custom(sampleGroup, sw.ElapsedMilliseconds);

   // Assert
   Assert.Less(sw.ElapsedMilliseconds, 2000,
     "Violation of OG_86650: " +
     "Levels should load under 2 seconds on the recommended configuration " +
     $"but \"{level.sceneName}\" loaded under {sw.ElapsedMilliseconds} " +
     "milliseconds");
}
```

Figure 19: The performance test that enforces the 2 seconds loading times across scenes

Figure 19 shows a typical custom performance test in Unity where not the Profile Markers are measured but a custom data called Load Time. For exact load time measurement stopwatch was used which is a high precision timer tool provided by the .NET library. Stopwatch is using the processor's internal timer therefore it is a perfect fit for performance measurements. The method loads and measures the loading time for every specified scene, this is provided by the `ValueSource` attribute which executes the test *n* times where *n* is the number of elements in the `_levels` variable. The measurement is recorded via the `Measure` class' static `Custom` method which samples the loading time only one time through the test and records it inside the supplied sample group. As we can see the test uses the AAA pattern as the comments indicate thus, the test can be clearly

separated into a setup (Arrange), Act and check results (Assertion) triad. The last interesting part about the code snippet in **Figure 19** is the assertion section. With a well formatted assertion text we can clearly indicate why the test phase failed. The assertion text even indicates the optimization goal that was violated in case of a long loading time. OG means optimization goal the number after it, is the ID of the story.

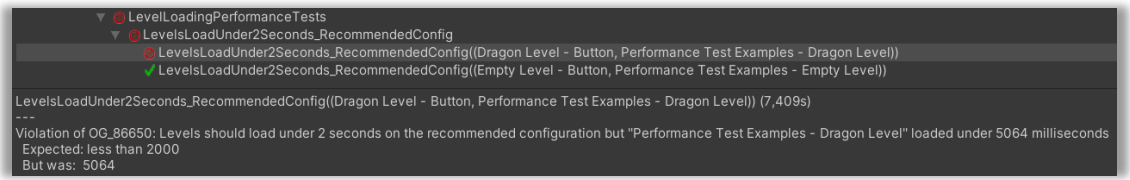

Figure 20: The result of the Dragon Level loading time test in case of a failure inside the Unity Test Runner's summary window.

5.2.2.2 Measuring frame times and FPS example

A common performance/optimization goal in game development is to guarantee a minimum FPS through the game. Fortunately, this does not mean that the frame time should be consistent but on average the FPS should be. For some strange reasons the Performance Test Extension does not provides an FPS counter for measurements, only frame time measurement through the Profiler API. In this example we will see what scoped measurements in PTE are and why they are so useful.

Scoped measurements in PTE are measurements that only sample between well-defined borders. In C# the most convenient way for scoping is the using block. Since PTE does not provide an FPS measurement extension the thesis' GitHub repo does. The `ScopedFPSMeasurement.cs` script contains a class that can be used for FPS measurements. Since the class implements the `Idisposable` interface the class can be used inside a using block allowing convenient scoped measurements.

```
using (ScopedFPSMeasurement.StartFPSMeasurement("FPS"))
{
    // Sampling will only happen here between the brackets
}
```

Figure 21: The advantage of scoped measurements

In this example, we assume the development team defined an optimization goal for the game itself which states that on the recommended configurations the game should run above 120 fps. Because of this, on computation heavy levels performance tests are

guarding this goal. One of this computation heavy levels is the Dragon Level. On this level the player can spawn quite many fire-balls and every fireball has a computation heavy material and a Rigidbody.

```
[UnityTest, Performance]
public IEnumerator SpawnFireBalls_RecommendedConfig_Min120FPS()
{
   // Arrange
   var fireBallSpawner =
      GameObject.Find("Spawner").GetComponent<FireBallSpawner>();
   // Small idling aka warmup before measurement starts
   yield return new WaitForSecondsRealtime(2.0f);
   // Simulating user input delay
   var userInputDelayYieldInstruction = new WaitForSecondsRealtime(0.15f);

   // Act
   using (Measure.Frames().Scope("Frame Time"))
   using (ScopedFPSMeasurement.StartFPSMeasurement("FPS"))
   {
       for (int i = 0; i < 250; i++)
       {
           fireBallSpawner.SpawnFireBalls(25);
           yield return userInputDelayYieldInstruction;
       }
   }

   // Assert
   // Calculating the results for assertions
   PerformanceTest.Active.CalculateStatisticalValues();
   var fpsResults =
      PerformanceTest.Active.SampleGroups.Find(s => s.Name == "FPS");

   Assert.GreaterOrEqual(fpsResults.Median, 120,
      "Violation of OG_117650: " +
      "The median FPS should be higher than 120 frames per second.");
}
```

Figure 22: The performance test for the dragon level

In order to maintain stable FPS with every changes the team wrote the performance test in **Figure 22** inside the `DragonLevelPerformanceTest.cs`. The above test loads the Dragon Level scene (cannot be seen because it is in the setup method of the class) then spawns fireballs with a simulated click delay. The frame time and FPS are only measured in the critical part where the computation heavy calculations happen. Via scoped measuring our test will be more precise since idling is not part of the sampling. After running the test, the Unity Test Report tool will give us a detailed summary about the results.

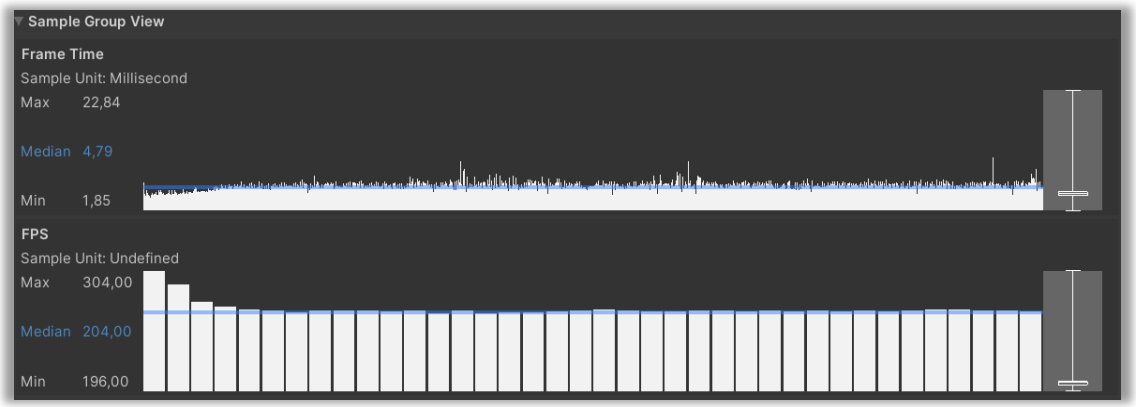

Figure 23: The results of the scoped measurements inside the Unity Test Reporter

We can see that the frame time fluctuates quite much however, the FPS is quite consistent since so many frames are rendered in a second this fluctuation disappears. Of course, if the test fails the results are still available and also the developers will get a notification that a recent change made the scene too computational heavy therefore violating the optimization goal.

5.3 Benchmarking

A common mistake when applying optimization to a component is not verifying that we are indeed using the optimal solution from the perspective of the goal. E.g. in the **2.3.2** (Pre-production) section we mentioned an example about choosing a proper physics solution for our project. If we only read the marketing description and the first page of the documentation of each physics solution, we might choose a physics solution that is not the most optimal for our use case. This is when benchmarking comes into play. In optimization, benchmarking is the process of comparing different solutions' performance via properly set up test cases. Via properly setting up benchmarks we can verify by exact numbers that indeed the chosen solution is the most optimal for our use-case. For benchmarking in Unity, we can use the frameworks, tools and packages mentioned in the previous section since they can give us detailed measurements both from the profiler and from a custom perspective. After the results are available we can compare them with one of the mentioned tools.

6 Scripting Optimization Techniques for Unity

Every game engine has a different architecture therefore there are some optimizations that need to be considered more in one than in the other. E.g. an Unreal Engine developer does not have to worry about interop calls while a Unity developer should. An Unreal Engine developer can opt-out from garbage collection while a Unity developer cannot and while an Unreal engine developer should worry about memory leaks a Unity engine developer usually should not. Because of the nature of these architectural differences there are many optimization techniques that only apply to game engines sharing similar architecture as Unity. In this section, we will discuss some of these advanced optimization techniques.

6.1 Update manager

The Unity engine has a Messaging system which allows the developers to define methods that will be called by an internal system based on their functionalities. One of the most commonly used Messages is the Update message which is called in every frame by the engine in order to Update the state of the GameObject. Every MonoBehaviour has this Message yet there is one strange thing about it which is that none of the classes in the inheritance hierarchy have a method defined by the name Update. Experienced .NET developers will probably think that if the method is not defined then Unity must be using reflection to find these Messages and call them. But the truth is that Unity is inspecting the script the first time the type is accessed (independently from the scripting backend) and checks if any of the Message methods are defined. If a Message method is defined then the engine will cache this information. If an instance of this type is instantiated then the engine will add it to the appropriate list and will call the method whenever it should. This is the key reason why Unity does not care about the visibility of our Message method.

```
public class Example1 : MonoBehaviour
{
    private void Update() { }
}
public class Example2: MonoBehaviour
{
    public void Update() { }
}
```

Figure 24: Both solutions will achieve the same results.

However, it is advised to give private visibility to our Message methods since they are supposed to be called only by the engine's internal messaging system. This mechanism is also the reason why the engine does not call these messages in a deterministic order.

The main problem with this approach is that every time the engine calls a Message method an interop call has to happen. We already talked about this at the end of section **4.1**. Fortunately, Messages rarely do marshalling so the overhead is smaller than in the example in section **4.1,** yet not negligible in some cases. E.g. if our game handles thousand or tens of thousands of objects which all have a script requiring a Message call then this overhead can be significant. A solution to this is to avoid interop calls. A good approach to this is behavior grouping. If we have a MonoBehaviour that is attached to a huge number of GameObjects we can cut the number of interop calls to just one by introducing an update manager. Since the update manager is also a managed object running managed code the only interop call will happen between the update manager's Update Message and the Unity engine's internal Message handler. We have to note the fact that this optimization technique is only relevant in large-scale projects, and the frame time saved via this technique is more impactful when using the Mono scripting backend. (Remember IL2CPP transpiles to C++).

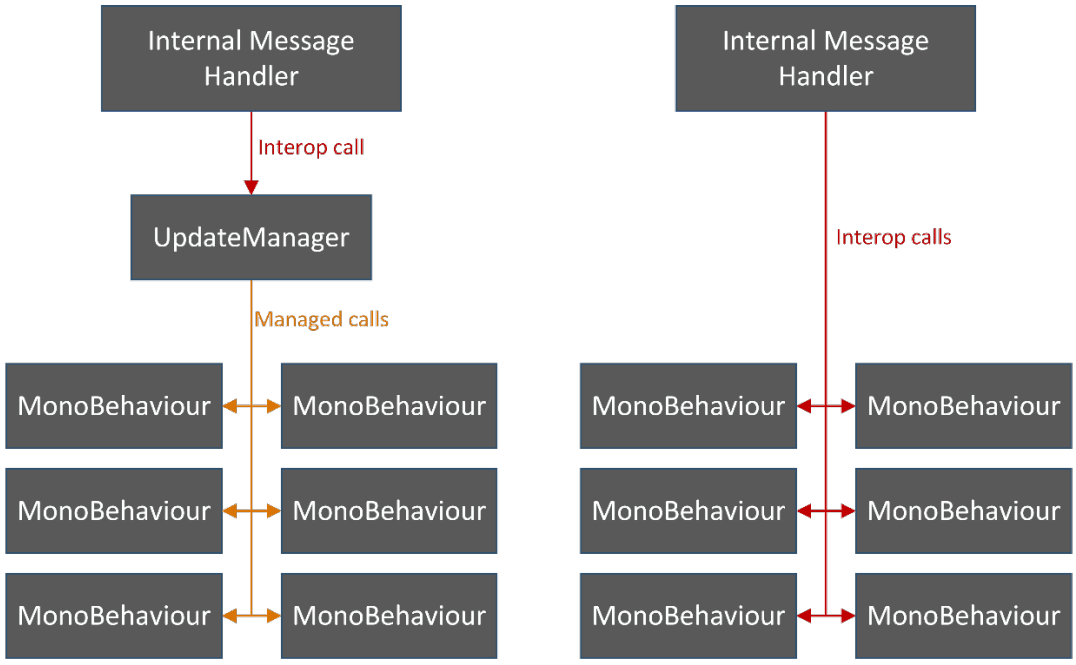

Figure 25: The concept of update managers in contrast to the traditional way.

After understanding **Figure 25** we can implement a basic update manager, then we will benchmark an example scene with and without an update manager. The example scene is named Update Manager. The folder and namespace structure shares the same properties as previous examples.

The Update Manager scene spawns 10 000 objects. All of these objects have a `Mover` script that slowly moves them up and down. The `Spawner` class that spawns the aforementioned objects has a `UseUpdateManager` flag which if at the initialization is set to true will use the `UpdateManager` class for updates otherwise will use the traditional internal Message handler.

```
public class UpdateManager
{

   static HashSet<ManagedMover> _updateables = new HashSet<ManagedMover>();

   public static void Add(ManagedMover mover)
   {
      _updateables.Add(mover);
   }

   public static void Remove(ManagedMover mover)
   {
      _updateables.Remove(mover);
   }

   class UpdateManagerInnerMonoBehaviour : MonoBehaviour
   {
      void Update()
      {
         foreach (var mover in _updateables)
         {
            mover.UpdateManager_Update();
         }
      }
   }

   #region Static constructor and field for inner MonoBehaviour
   ...
   #endregion

}
```

Figure 26: The UpdateManager class

As we can see both the concept and the implementation is simple. Of course, this simple example only allows the tracking of `ManagedMover` types but this does not mean we are only able to track one type at a time. We have plenty of tools to fix the problem (e.g. interfaces) but for the sake of simplicity and shortness we will go with the above method. The only drawback of the `UpdateManager` is that from now on, it is our

responsibility to remove the `ManagedMover` instances from the `_updateables` collection while in the traditional way Unity would handle this for the developers. Luckily, we can solve this easily with `OnDisable` and `OnEnable` Messages as it can be seen on **Figure 27**.

```
void OnEnable() => UpdateManager.Add(this);

void OnDisable() => UpdateManager.Remove(this);
```

Figure 27: Simply calling `OnEnable` and `OnDisable` will do the trick for automatic state management for custom updatable `MonoBehaviours`.

6.1.1 Results

Benchmarking a scenario where the rendering is not relevant should happen without a camera and rendering set to low wherever possible. This is important to keep in mind especially when our goal is to get extremely accurate results. Since our scene will do the exact thing rendering wise no matter of the update solution we use, for the sake of visual experience the camera will not be turned off.

Before writing the benchmark, we should plan how we should execute it. Since a test scene is available, the most straightforward solution is to use this test scene and measure its performance. In the previous sections we talked about the different scripting setups available in Unity so this should be taken into consideration as well since some of the platforms support one another. If the benchmark is about telling the real difference between the two solutions the measurement should be restricted to the relevant parts. So, measuring the frame time and FPS here is not the best benchmarking strategy here therefore we are going to use a `StopWatch` and measure the relevant parts only.

Measuring the `UpdateManager` is easy. We just put a `StopWatch` before and after the `foreach` block and sample it every time via a delegate inside the benchmark.

```
void Update()
{
    SW.Restart();
    foreach (var mover in _updateables)
    {
       mover.UpdateManager_Update();
    }
    SW.Stop();
    StopWatchStoppedCallback?.Invoke();
}
```

Figure 28: The added `StopWatch` to the `UpdateManager`'s `Update` Message.

On the other hand, measuring the simple `Update` messages is not that trivial. Since so many game loops are executed under a second, we do not want to pollute our `Update` method so the measurement and checking must be moved elsewhere. A good solution to this problem is to create two `MonoBehaviours` and set their execution order just before and after the `Mover` script. The only drawback of this solution is that we will have an additional interop call in order to stop the timer in the just after MonoBehaviour. The two helper classes are named UpdateBenchmarkHelperJustBefore and After and located in the Test assembly. The stopping and measuring logic are the same as in **Figure 28**.

Figure 29: The results of the benchmarks under different scripting backends

Not surprisingly IL2CPP leading the competition by far however it's still interesting that the Update Manager is still twice as fast as the traditional way. If we profiled the execution of the Traditional method's IL2CPP build we would probably find many Unity specific calls like check if the GameObject exists before invoking a component method etc. and these would probably explain the longer execution time. We can also make the conclusion that IL2CPP in this case is far faster than Mono, usually around twice as fast. The benchmark ran for one minute prior to a 5 seconds warmup and both scripting backends had the ideal compiler setting.

6.2 Custom properties and hashing

Unity has many internal systems which allow the developers to create own properties in order to extend those systems. Systems and modules like the Animator or Material and Shader properties are all allow this type of behavior. Whenever the developer creates such a property a unique name has to be assigned to it. However, Unity does not use this unique string name to address these objects. Under the hood a hash is calculated based on the string value and the addressing will happen via this unique ID. If the developer tries to assign a non-unique name to a property Unity will automatically concatenate a unique number after the name thus enforcing uniqueness.

The problem with this is that many developers do not know about this mechanism and tend to access these properties using the overload methods that are accepting a string value. In this case, the method calculates the hash and then assigns or returns the desired value. Normally this does not produce measurable differences in frame time since it's quite rare that for example the Animator's state is queried in every frame. Howbeit this can cause performance degradations in cases when it is used e.g. in tight loops or in Update Messages. A good example to this are shaders. It is a common scenario where a shader has to be updated via a C# script in order to execute some kind of special culling, calculate light based on game logic etc. Fortunately, Unity provided methods that are directly accepting hashed values for performance improvements. So, a proper solution to avoid hash calculations every time the get/set property value methods are invoked is to pre-calculate these hashes at application startup or at scene load and use the overload methods that are directly consuming these hashes.

When working with systems that allow the definition of custom properties, a good design pattern proposed by this thesis is to use a mapping class containing the precalculated hash values via the advantage of static classes. In C#, static classes are a type of classes that cannot be instantiated and therefore can only contain static members. Since a static class cannot be instantiated inheritance is also prohibited. Moreover, the application cannot specify exactly when the class is loaded. However, it is guaranteed to be loaded and to have its fields initialized before the class is referenced for the first time in the program. A static class remains in memory for the lifetime of the application. Static classes are therefore a perfect choice for storing "global" state about the application. Since all of the hashing methods are static ones, we can use them in static constructors or in static field initializers.

The concept of the proposed design pattern is that whenever a shader is created that needs input or the game needs output from it, a corresponding static mapping class with the same name as the shader has to be created based on **Figure 30**.

```
namespace ExampleProject.Shaders
{
   public static class StylizedWater
   {
      public static readonly int _Color =
         Shader.PropertyToID(nameof(_Color));

      public static readonly int _FogColor =
         Shader.PropertyToID(nameof(_FogColor));

      public static readonly int _IntersectionColor =
         Shader.PropertyToID(nameof(_IntersectionColor));
   }
}
```

Figure 30: The proposed design pattern for storing pre-calculated hash values for shaders

In **Figure 30** we can see a hash mapping static class for the `StylizedWater` shader. The `StylizedWater` shader needs three input parameters from the game thus the mapping class has three int values containing the pre-calculated hash values with the same name as in the `StylizedWater` shader. By defining the properties with the same name, we can avoid the use of magic strings since with the `nameof` keyword we can convert a variable name to a string value. Therefore, whenever we need to change the property's name inside the shader, we only need to change the variable name in the static class which can be conveniently done via IDE specific refactor tools. Moreover, all of the mapping values are `readonly` so accidental value assignments are prohibited by the compiler.

```
Shader "Custom/StylizedWater"
{
   ...
   fixed4 _Color;
   fixed4 _FogColor;
   fixed4 _IntersectionColor;
   ...
}
```

Figure 31: The parameter declarations of the StylizedWater shader

```
_material.SetColor(StylizedWater._Color, Color.cyan);
```

Figure 32: Example usage of the proposed design pattern

6.2.1 Results

The example scene for the benchmarking is called Importance of Hashing. The scene was built on the example scene of Harry Alisavakis [24] showing his stylized water shader. The optimization is scripting backend independent so IL2CPP was used since that is the recommended backend for standalone windows.

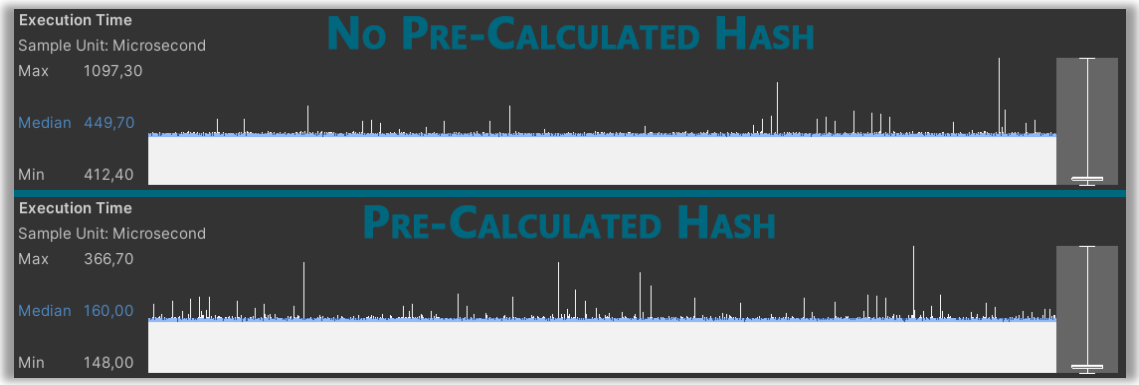

Figure 33: The results of the benchmarks

As it can be seen on **Figure 33** this optimization is a micro optimization and on its own won't change that much since the measurement unit now is in micro, not in milliseconds. However, in large projects and on low end devices this might be a nice optimization on its own. Pre-calculated hashing is also extremely important on consoles since on PlayStation and on Xbox most of the graphics touches are achieved via proper shading rather than on model complexity.

The benchmark ran for one minute prior to a 5 seconds warmup. For more impactful numbers 250 shaders were used meaning that in each frame 1000 hash values were calculated with the unoptimized solution rather than just a value assignment/retrieval.

Since this optimization technique is more impactful on lower end devices the benchmarks were repeated on an Android phone with the following hardware.


```
"Hardware": {
  "OperatingSystem": "Android OS 8.1.0/API-27(OPM1/15.2016.1902.340-20190216)",
  "DeviceModel": "asus ASUS_X00TD",
  "DeviceName": "ASUS_X00TD",
  "ProcessorType": "ARM64 FP ASIMD AES",
  "ProcessorCount": 8,
  "GraphicsDeviceName": "Adreno (TM) 512",
  "SystemMemorySizeMB": 2744
}
```


Figure 34: The hardware info of an Asus Zenfone Max Pro M1

Since we cannot setup the same performance requirements for a high-end desktop PC and for a low-end Android device only 20 shaders were used in **Figure 35**.

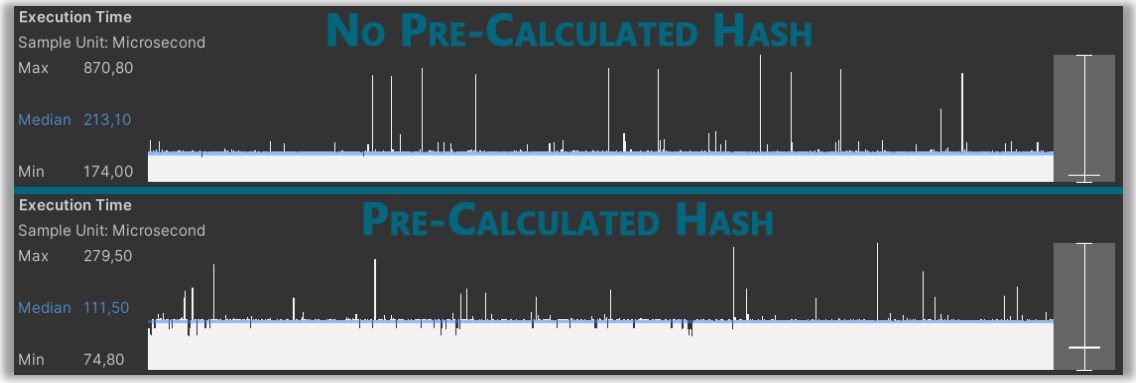

Figure 35: The results of the lighter benchmarks on an Asus Zenfone Max Pro M1

Note the outlier values! Under 60 seconds around 10 frames suffered nearly a millisecond stuttering because of hash value calculations. In a performance critical mobile game where graphics are important and animations are highly used this is unacceptable. The tests were run ten consecutive times and in all of them the outlier values were overrepresented. In order to visualize how much stuttering happens over a one-minute time span, the results were ordered from highest to lowest execution time (**Figure 36**).

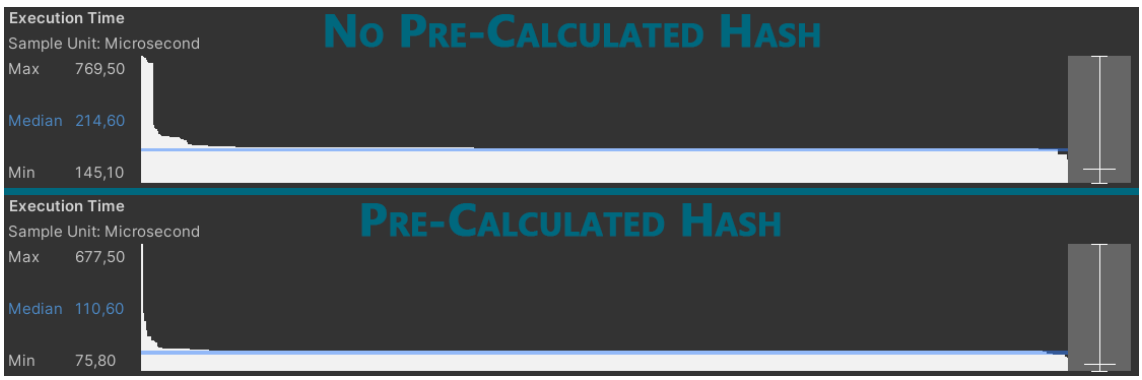

Figure 36: The 10th consecutive run on the Android phone in decreasing order.

Note how the pre-calculated hash has only one serious outstanding value while the no pre-calculated hash solution has 10-20 times more. To sum it all, this is a micro optimization. Yet such a simple optimization technique that should be always applied. Moreover, the proposed design pattern avoids the usage of magic strings thus making the code more readable and easier to refactor.

6.3 Struct assignment

When a project depends on the mono runtime for some reasons and the project also heavily utilizes vector math or struct creation in general, there is an optimization that can save serious micro or milliseconds. Whenever a struct is created via the `new` keyword the constructor is called thus a stack frame allocation happens. The parameters and some meta data are pushed onto the stack and when the constructor is finished all the relevant data has to be deallocated. In case of large data, this can mean that serious micro or milliseconds are spent on constructor function calls. This is the case with vector math in Unity. Unity implemented scaling in the following way **Figure 37**.

```
public static Vector3 operator*(Vector3 a, float d)
{
    return new Vector3(a.x * d, a.y * d, a.z * d);
}
```

Figure 37: Implementation of scaling in Unity 2019 LTS

We can see that when vector * scalar returns it calls the constructor of `Vector3`. A more optimal solution when using the mono runtime would be the one in **Figure 38**.

```
public static Vector3 operator*(Vector3 a, float d)
{
    Vector3 result;
    result.x = a.x * d;
    result.y = a.y * d;
    result.z = a.z * d;
    return result;
}
```

Figure 38: A more optimal solution when using the mono backend

The key move here is that we do not call the struct's constructor but we allocate the memory needed for it with empty values. After this we just assign the individual components and return this value. This way we can avoid one more indirection via a constructor function call and additionally we do not have to allocate and deallocate on the stack.

6.3.1 Results

Let's see some benchmarks (**Figure 39**) using the mono backend on the PC configuration.

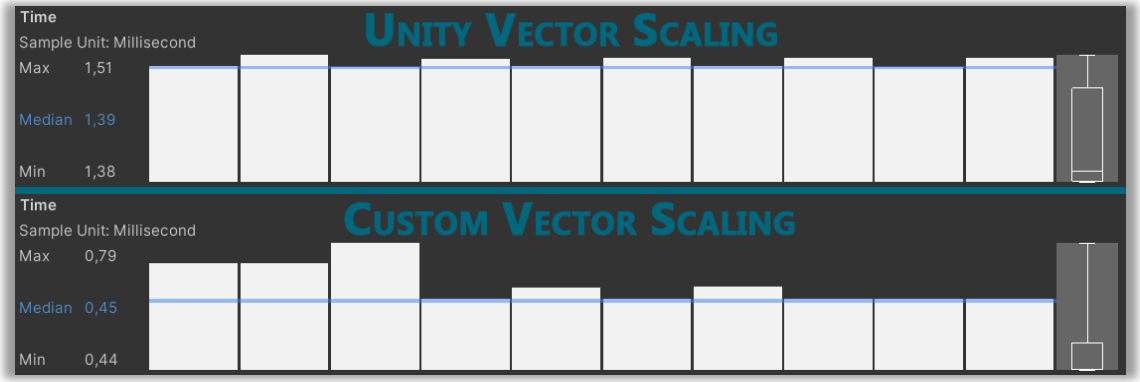

Figure 39: The difference can be quite impactful when using large data with Mono

The benchmark is doing 50 000 vector scaling with 10 warmup iterations. It is located inside the `VectorAssignmentBenchmark.cs` file.

But if this is such a problem why Unity does not fix it?

The answer is simple. Mono is not the recommended scripting backend for production builds. The truth is that the previous optimization is not just useless but actually harmful on an IL2CPP build as it can be seen in **Figure 40**. Therefore, Unity does not intend to slow down its flagship backend obviously. The only interesting question is what would Unity do if they are planning to switch Mono to CoreCLR when .NET 6 is introduced. But that's a question for the future.

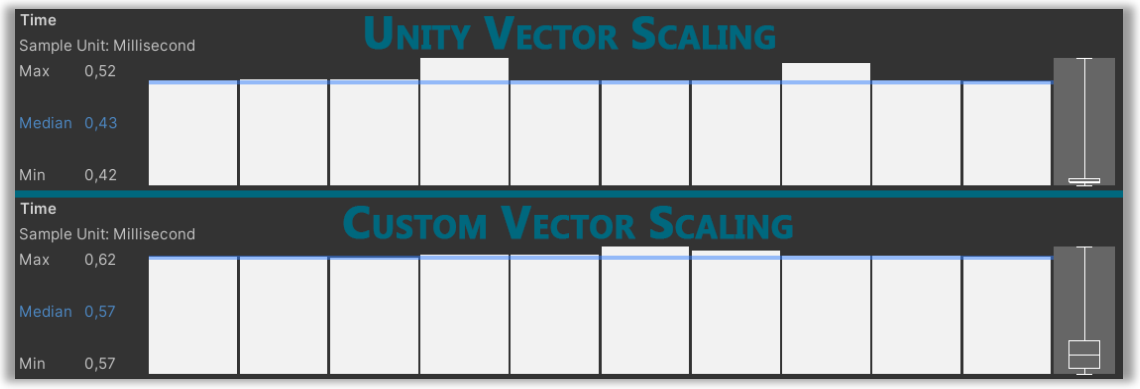

Figure 40: An IL2CPP build is actually slower when this optimization is applied

6.4 Hierarchy Optimization

In Unity every GameObject has a Transform component. It cannot be removed and only one can be attached to a single GameObject. Every time a GameObject moves, altering its transform, we need to let every game system that might care about this know. Rendering, physics, and every parent and child of the GameObject needs to be informed of this movement in order to match. As the project grows in size a heavily composited hierarchy could spend serious micro or milliseconds sending and receiving Transform Changed Messages. So, while ordering our GameObjects logically can really help navigating in the hierarchy, over organizing can cause serious performance penalties. It's extremely hard to present an example where this can cause serious performance penalty since the project has to be a large one, however it can be still measured on smaller projects albeit with more modest results.

In **Figure 41** we can see a heavily composited hierarchy or we could say an over organized object hierarchy.

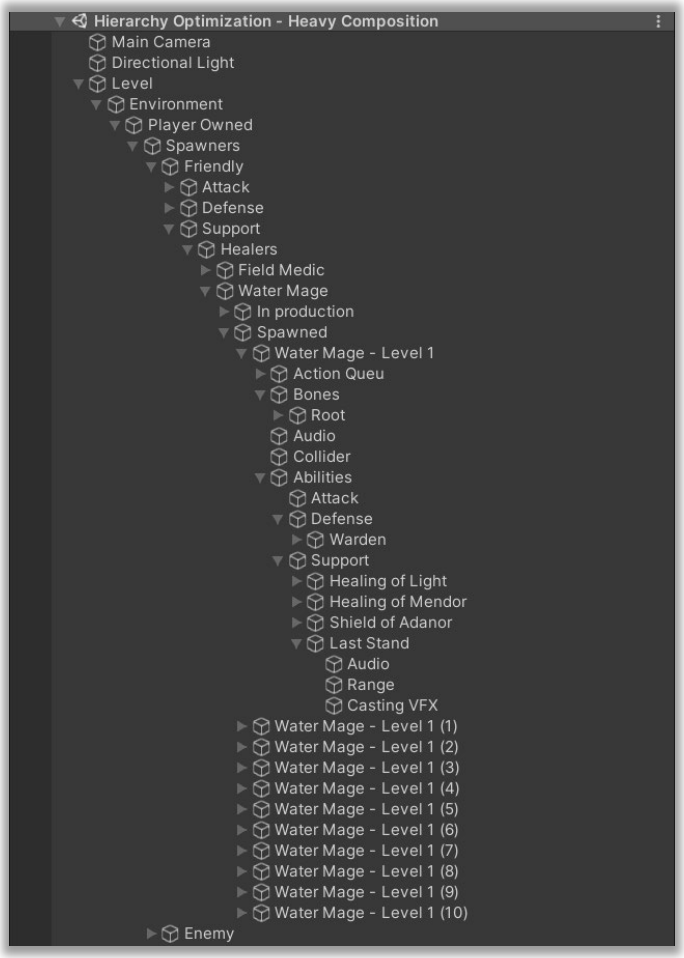

Figure 41: An over organized object hierarchy

As we can see when a unit is spawned it is grouped under a spawner GameObject but a spawner object is grouped under several other GameObjects as well. The Water Mage unit type uses an over organized hierarchy where audio, VFX and abilities also have dedicated GameObjects thus dedicated transforms. This would not be a problem if the hierarchy tree would be static but “unfortunately” mages can move or new units can be spawned. Whenever a mage moves, the whole hierarchy tree have to be updated via Transform Changed Messages. The script which moves the mage is located on the mage’s root object (Water Mage – Level 1).

Let’s optimize this over organized hierarchy for performance reasons. Firstly, when moving the mage, we do not have to move all its abilities audio files etc. So, the first optimization would be to move the mover script to the root bones GameObject. This way only the relevant parts are moved, however we are still sending countless Transform Changed Messages towards parent GameObjects. Even though grouping the spawned units under the respective category is a good organization strategy, if we leave the editor scope it becomes useless. Removing this deep composition and spawning the units on the top level or under a common shallow composited container GameObject will greatly improve our performance.

6.4.1 Results

Let’s see a benchmark where we move 300 mages in an empty scene. The scene used by the benchmark is called Hierarchy Optimization – Heavy Composition. The benchmark will run for 60 seconds prior to a 5 seconds warmup phase.

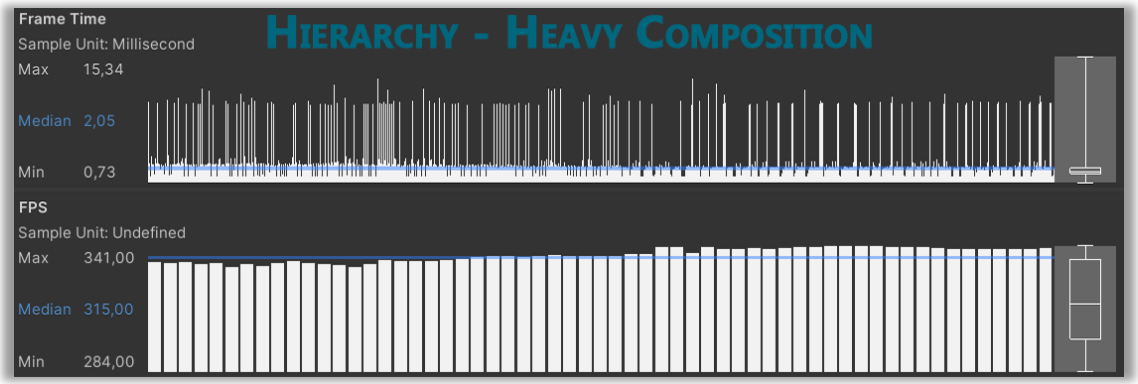

Figure 42: The benchmark results of the heavy composition

Figure 42 on its own does not tell much so let's see a benchmark with the applied changes where the units are spawned on top level. The scene used by the benchmark is called Hierarchy Optimization – Optimized Composition.

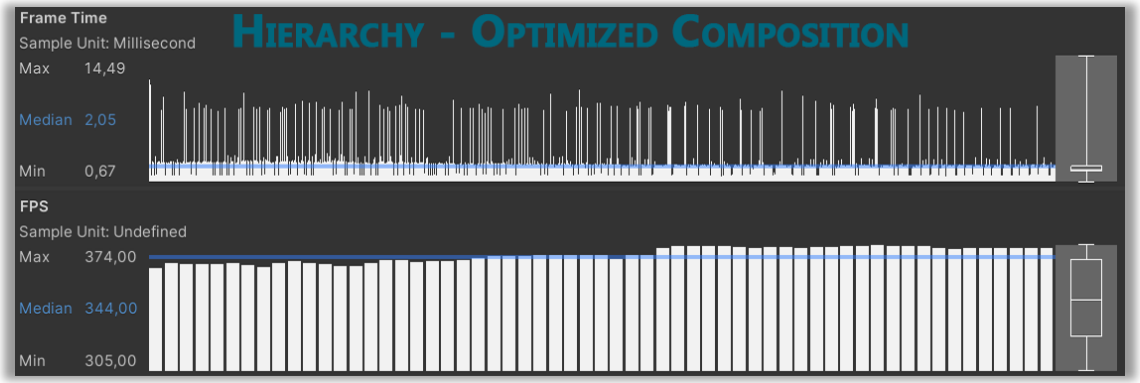

Figure 43: The benchmark results of the optimized hierarchy

As we can see via the small applied changes in this small and simple scene allowed us to boost our FPS by 10%.

7 Conclusion

In this thesis we explored the field of optimization in game development. We learned how to identify possible performance critical areas of our system early in the development process. We learned how to handle arising performance degradations via setting up proper optimization goals in order to handle these degradations from observation to implementation. We mastered when and what type of optimizations we should do in the different stages of the development process. Afterwards, we took a deep dive into the mechanisms of the popular real-time development platform Unity. Investigated why Unity choose C# as their scripting language and how it is used by both Mono and the IL2CPP scripting backend. After this, we gained knowledge about the available tools and frameworks for performance analysis in Unity. We proposed a general solution for automatic performance degradation detection and learned about why benchmarking and performance testing is crucial in game development as well. At the end of this thesis, after mastering the aforementioned topics, we gathered some advanced optimization techniques and illustrated them via real life scenarios. Finally, we benchmarked their results through proper performance tests.

Bibliography

[1] KNUTH, Donald E. Structured programming with go to statements. *ACM Computing Surveys (CSUR)*, 1974, 6.4: 261-301.

[2] Unity Technologies, *Our Impact by the numbers*
https://unity.com/our-company
Acessed 2020.09.20

[3] Steam, *The First Tree*
https://store.steampowered.com/app/555150/The_First_Tree/
Acessed 2020.09.28

[4] Learning Hub, *The 7 Stages of Game Development*,
https://learn.g2.com/stages-of-game-development
Acessed 2020.10.02

[5] Unity Technologies, *Unparalleled platform support*
https://unity.com/features/multiplatform
Acessed 2020.10.06

[6] Wilmer Lin, *ECS: Performance by Default*
https://www.raywenderlich.com/7630142-entity-component-system-for-unity-getting-started#toc-anchor-003
Acessed 2020.10.07

[7] Unity Technologies, *DOTS Packages*
https://unity.com/dots/packages
Acessed 2020.10.07

[8] Unity Technologies, *Unity Reference-Only License*
https://unity3d.com/legal/licenses/Unity_Reference_Only_License
Acessed 2020.10.10

[9] Microsoft Docs, *C# Documentation*
https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/
Acessed 2020.10.11.

[10] Wikipedia: *C Sharp (programming language)*,
https://en.wikipedia.org/wiki/C_Sharp_(programming_language)
revision 11:53, 11 October 2020

[11] Microsoft Docs, *What is "managed code"?*
https://docs.microsoft.com/en-us/dotnet/standard/managed-code
Acessed 2020.10.14

[12] Microsoft Docs, *Common Language Runtime (CLR) overview*
https://docs.microsoft.com/en-us/dotnet/standard/clr
Acessed 2020.10.16

[13] ZDNet, *Security bugs and memory leaks*
https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/
https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/
Acessed 2020.10.16

[14] Eric Lippert, *Mutating Readonly Structs*
https://docs.microsoft.com/en-us/archive/blogs/ericlippert/mutating-readonly-structs
Acessed 2020.10.26

[15] Microsoft Docs, *Calling Native Functions from Managed Code*
https://docs.microsoft.com/en-us/cpp/dotnet/calling-native-functions-from-managed-code?redirectedfrom=MSDN&view=msvc-160
Acessed 2020.10.28

[16] Wikipedia: *Mono (software)*
https://en.wikipedia.org/wiki/Mono_(software)
revision 07:02, 3 November 2020

[17] Alexandre Mutel, *Porting the Unity Engine to .NET CoreCLR*
https://xoofx.com/blog/2018/04/06/porting-unity-to-coreclr/
Acessed 2020.11.08

[18] Josh Peterson, *An introduction to IL2CPP internals*
https://blogs.unity3d.com/2015/05/06/an-introduction-to-ilcpp-internals/
Acessed 2020.11.10

[19] Wikipedia: *Profiling (computer programming)*
https://en.wikipedia.org/wiki/Profiling_(computer_programming)
revision 06:01, 19 November 2020

[20] Unity Manual, *Profiler overview*
https://docs.unity3d.com/Manual/Profiler.html
Acessed 2020.11.19

[21] Unity Manual, *Performance Testing Extension for Unity Test Runner*
https://docs.unity3d.com/Packages/com.unity.test-framework.performance@latest/index.html
Acessed 2020.11.20

[22] Sean Stolberg, Performance Benchmarking in Unity: How to Get Started
https://blogs.unity3d.com/2018/09/25/performance-benchmarking-in-unity-how-to-get-started/
Acessed 2020.11.21

[23] Valentin Simonov, *10000 Update() calls*
https://blogs.unity3d.com/2015/12/23/1k-update-calls/
Acessed 2020.11.25

[24] Harry Alisavakis, *Stylized water shading*
https://halisavakis.com/my-take-on-shaders-stylized-water-shader/
Acessed 2020.11.28

[25] William Armstrong, Optimizing the Hierarchy
https://blogs.unity3d.com/2017/06/29/best-practices-from-the-spotlight-team-optimizing-the-hierarchy/
Acessed 2020.11.30.