FELADATKIÍRÁS

A feladatkiírást a **tanszék saját előírása szerint** vagy a tanszéki adminisztrációban lehet átvenni, és a tanszéki pecséttel ellátott, a tanszékvezető által aláírt lapot kell belefűzni a leadott munkába, vagy a tanszékvezető által elektronikusan jóváhagyott feladatkiírást kell a Diplomaterv Portálról letölteni és a leadott munkába belefűzni (ezen oldal HELYETT, ez az oldal csak útmutatás). Az elektronikusan feltöltött dolgozatban már nem kell megismételni a feladatkiírást.



Budapest University of Technology and Economics

Faculty of Electrical Engineering and Informatics

Menyhárt Bence

Game engine specific optimization techniques for Unity

Consultant

Milán

BUDAPEST, 2020

Contents (frissítsd majd az oldalszámokat)

[Összefoglaló 5](#_Toc332798843)

[Abstract 6](#_Toc332798844)

[1 Introduction Hiba! A könyvjelző nem létezik.](#_Toc332798845)

[1.1 Formázási tudnivalók **Hiba! A könyvjelző nem létezik.**](#_Toc332798846)

[1.1.1 Címsorok **Hiba! A könyvjelző nem létezik.**](#_Toc332798847)

[1.1.2 Képek **Hiba! A könyvjelző nem létezik.**](#_Toc332798848)

[1.1.3 Kódrészletek **Hiba! A könyvjelző nem létezik.**](#_Toc332798849)

[1.1.4 Irodalomjegyzék **Hiba! A könyvjelző nem létezik.**](#_Toc332798850)

[2 Utolsó simítások 20](#_Toc332798851)

[Irodalomjegyzék 37](#_Toc332798852)

[Függelék 39](#_Toc332798853)

Hallgatói nyilatkozat

Alulírott **Menyhárt Bence**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot/ diplomatervet (nem kívánt törlendő) 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, 2020. 11. 20.

...…………………………………………….

Összefoglaló

Ide jön a ½-1 oldalas magyar nyelvű összefoglaló, melynek szövege a Diplomaterv Portálra külön is feltöltésre kerül.

Abstract

Ide jön a ½-1 oldalas angol nyelvű összefoglaló, amelynek szövege a Diplomaterv Portálra külön is feltöltésre kerül.

# 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 are 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 kind of disappointed after realizing that outside of a 150 unit range shadows would be culled. So, we the programmers are 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 trying 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 are these 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.

We are going to look into topics like:

* How and when in the development process we should define our target hardware and optimization goals
* Deep dive into how Unity works, especially its scripting backend. Identify which parts of it are usually performance critical
* Show how to properly benchmark Unity applications and what kind of tools are available for performance analysis
* Examine Unity specific optimizations and understand why they perform better

# 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 is the problem that can cause this freeze. This is done by profiling and measurements. The developers attach a profiling tool to the game so they can get results for CPU, memory, renderer, audio and storage usage. After analyzing the results, they can then identify which parts of the software causes 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.

## 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]](#footnote-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]](#footnote-2) (**Figure 1**) and assigned a priority to it. The Acceptance criteria[[3]](#footnote-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.

## 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 miss interpreted 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 connects it with the most of the time right fact that we should implement first then optimize our code (which I would correct to implement, measure and optimize anyway), so any thought about optimization before the implementation makes 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 possibly 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.

## 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 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 can not 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 pre-specify.

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



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

Each of this phase contains various topics that needs to be “answered”, some of them can vary from project to project but some of them are project independent.

### 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 just usually 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 and more. 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 what type of pre-defined optimization goals we might need, they are typically collected in the next stage of the pipeline.

### 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]](#footnote-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 they are sold but through IAP[[5]](#footnote-5), advertises and small content DLCs[[6]](#footnote-6).

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 into the game design document. We already know the target platforms and the desired gameplay mechanics so the engineers can collect the optimization goals based on this information. Moreover, as mentioned above the team should make a prototype to verify the gameplay, which can be also helpful to identify possible future bottlenecks.

When defining optimization goals, 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, physics and one of the main goal 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 60 frames per seconds on average
* 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]](#footnote-7) version
* The built-in 2D physics based on Box2D[[8]](#footnote-8) engine
* Unity physics, a complete deterministic rigid body dynamics and spatial query system written entirely in high performance C# using DOTS[[9]](#footnote-9)
* Havok Physics, an implementation of the Havok physics[[10]](#footnote-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.

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 is the exact problem 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 to 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]](#footnote-11) opposite 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

### 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 to coordinate the work of all these awesome and talented people

and we did not mentioned 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, this is the phase where we usually trying 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.

### 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 [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]](#footnote-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

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]](#footnote-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”.

# Unity

## 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, 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]](#footnote-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
  + Animation – An integrated animation workflow for both 2D and 3D
  + Cinemachine – Award winning real-time codeless camera behavior
  + Timeline – A tool to create cinematic content and complex particle effects
* Audio and video tools
* Asset store
  + An official store for sharing user created assets like models, animations, audios etc.
* VR[[15]](#footnote-15), MR[[16]](#footnote-16) and AR[[17]](#footnote-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
  + Selectable Scripting backend – Mono, IL2CPP
  + Unity Profiler – A profiler tool for performance analyzation
  + Frame Debugger – Debug the individual draw calls that are used to render a frame
  + Detailed API Documentation
* Rendering
  + Various Render Pipelines – Different render pipelines for different requirements
  + Visual Shader Editor
  + Visual VFX Editor
  + Post-processing
  + Shader skeletons and pre-built shaders
  + Ray tracing
  + Particle System
* Multiplayer and Networking
* Navigation and Pathfinding
* Physics 3D
  + Various Physics solutions
* Physics 2D
  + Box2D – A highly efficient 2D physics solution
* User Interface
  + Various UI solutions for both in-editor and in-game UI scripting
  + Visual UI Editor – A CSS like UI workflow called USS
* 2D
  + Sprite editing
  + Sprite Atlas
  + Tilemap support
  + 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
  + LOD[[18]](#footnote-18) system

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[[19]](#footnote-19), 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[[20]](#footnote-20). 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[[21]](#footnote-21) have to contain the Unity logo.   
In this thesis we will use the Pro version of Unity.

## 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 a data‑oriented design view over 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 just only the position of the zombie but it contains 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 Unity does since 2018 is that they are rewriting the core of the engine to harvest the full power of data-oriented programming. However, even though 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]

# 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 happens 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 part of our scripting logic. Unity’s source code is also proprietary which means 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[[22]](#footnote-22) of their .NET assemblies under a reference-only license[[23]](#footnote-23) 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 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.

## C# the scripting language of Unity

C# is a modern general-purpose, multi-paradigm[[24]](#footnote-24) 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[[25]](#footnote-25)/ECMA[[26]](#footnote-26)) 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 makes 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.

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, 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 type of bugs as well. C# also has other memory related security features like bounds 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 problem 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[[27]](#footnote-27)). 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.)

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 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 FPS game made with a GC backed language runs at 144 fps[[28]](#footnote-28) then suddenly the fps drops to 72. This is a quite common problem and usually indicates that the game allocates way too many objects on the heap[[29]](#footnote-29). Everything that gets allocated on the heap the GC will take care of. This means that if we allocate too many objects on the heap the GC will have to run frequently thus dropping our frame rate. Luckily C# can help us avoid unnecessary heap allocations in various ways. C# unlike Java distinguishes two main types from the point of the memory. Reference types and value types. While reference types are always allocated on the heap[[30]](#footnote-30) value types are allocated on the stack[[31]](#footnote-31) with some exceptions. The two most important exceptions when a value type is actually allocated on the heap are

* The value type is a field of a class
* The value type is boxed[[32]](#footnote-32)

But why is it good that value types are stored on the stack rather than on the managed heap? Because of efficiency. Deallocating the stack is extremely cheap compared to deallocations on the managed heap. Moreover, types allocated on the stack are no subject to the GC. Since value types 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[[33]](#footnote-33). Luckily C# supports structs that are happens to be value types that supporting 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 scenario 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 would be 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 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 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’s implementation 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 provides all the necessary resources for cross-language support. C# supports interoperability with language level syntaxes. The framework provides marshalling[[34]](#footnote-34) 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.

Although interop calls are extremely useful and they are quite crucial in game development they do have a significant overhead compared to simple method call.  
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[[35]](#footnote-35) 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.

## .NET in Unity

.NET is a developer platform made up of different kind 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. It is released under a MIT[[36]](#footnote-36) License.

* **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.

# Utolsó simítások

Miután elkészültünk a dokumentációval, ne felejtsük el a következő lépéseket:

* Kereszthivatkozások frissítése: miután kijelöltük a teljes szöveget (Ctrl+A), nyomjuk meg az F9 billentyűt, és a Word frissíti az összes kereszthivatkozást. Ilyenkor ellenőrizzük, hogy nem jelent-e meg valahol a "Hiba! A könyvjelző nem létezik." szöveg.
* Dokumentum tulajdonságok megadása: a dokumentumhoz tartozó meta adatok kitöltése (szerző, cím, kulcsszavak stb.). Erre való a Dokumentum tulajdonságai panel, mely a Fájl / Információ / Tulajdonságok / Dokumentumpanel megjelenítése úton érhető el.
* Kinézet ellenőrzése PDF-ben: a legjobb teszt a végén, ha PDF-et készítünk a dokumentumból, és azt leellenőrizzük.
* Kérdezd meg mi van akkor ha nem nincs úgy általános hviatkozás az irodalomjegyzékre
* Kérdezd meg kép az függelékhez kerül inkább

Irodalomjegyzék

1. Donald Knuth,   
   Structured Programming With Go To Statements, *Premature Optimization*  
   <https://proxy.c2.com/cgi/wiki?PrematureOptimization>
2. Unity Technologies, *Our Impact by the numbers*  
   <https://unity.com/our-company>
3. Steam, *The First Tree* <https://store.steampowered.com/app/555150/The_First_Tree/>
4. Learning Hub, *The 7 Stages of Game Development*,  
   <https://learn.g2.com/stages-of-game-development>
5. Unity Technologies, *Unparalleled platform support*  
   <https://unity.com/features/multiplatform>
6. Wilmer Lin, *ECS: Performance by Default*  
   [https://www.raywenderlich.com/7630142-entity-component-system-for-unity-getting-started#toc-anchor-003](https://www.raywenderlich.com/7630142-entity-component-system-for-unity-getting-started%23toc-anchor-003)
7. Unity Technologies, *DOTS Packages*  
   <https://unity.com/dots/packages>
8. Unity Technologies, *Unity Reference-Only License*  
   <https://unity3d.com/legal/licenses/Unity_Reference_Only_License>
9. Microsoft Docs, *C# Documentation*  
   <https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/>
10. Wikipedia: C Sharp (programming language), <https://en.wikipedia.org/wiki/C_Sharp_(programming_language)>  
    (revision 11:53, 11 November 2020)
11. Microsoft Docs, *What is “managed code”?*  
    <https://docs.microsoft.com/en-us/dotnet/standard/managed-code>
12. Microsoft Docs, *Common Language Runtime (CLR) overview*  
    <https://docs.microsoft.com/en-us/dotnet/standard/clr>
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/>
14. *Eric Lippert*, Mutating Readonly Structs  
    <https://docs.microsoft.com/en-us/archive/blogs/ericlippert/mutating-readonly-structs>
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>
16. Wikipedia: Mono (software)  
    <https://en.wikipedia.org/wiki/Mono_(software)>  
    (revision 07:02, 3 November 2020)

Függelék

1. A list of stories that needs to be done by the team. They are usually ordered by priority. [↑](#footnote-ref-1)
2. Represents a feature that needs to be done in agile development [↑](#footnote-ref-2)
3. A story can not be marked as verified till it does not fulfill all the requirements listed in it [↑](#footnote-ref-3)
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. [↑](#footnote-ref-4)
5. **I**n-**a**pp **p**urchases is a common way to monetize free to play games [↑](#footnote-ref-5)
6. **D**own**l**oadable **c**ontent is an expansion to a game in order to extend it with more playable content. [↑](#footnote-ref-6)
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. [↑](#footnote-ref-7)
8. Box2D is an open source 2D physics engine written by Erin Catto [↑](#footnote-ref-8)
9. **D**ata-**O**riented **T**echnology **S**tack, a solution by Unity to take advantage of modern multicore processors [↑](#footnote-ref-9)
10. Havok physics is an industry leading complete physics solution for games and modelling software [↑](#footnote-ref-10)
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 [↑](#footnote-ref-11)
12. **Q**uality **o**f **L**ife improvements usually make the gameplay more ergonomic and user-friendly [↑](#footnote-ref-12)
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 [↑](#footnote-ref-13)
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 [↑](#footnote-ref-14)
15. **V**irtual **R**eality is a type of application which simulates an environment around the user [↑](#footnote-ref-15)
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 [↑](#footnote-ref-16)
17. **A**ugmented **R**eality is a type of application which projects part of the digital world over the real world [↑](#footnote-ref-17)
18. **L**evel **o**f **D**etail is a common solution to reduce graphical complexity of objects further away from the camera [↑](#footnote-ref-18)
19. A type of change in a software system that potentially causes other parts of the system to fail. [↑](#footnote-ref-19)
20. [A general comparision of Unity licenses](https://store.unity.com/compare-plans) [↑](#footnote-ref-20)
21. The screen that is shown when the Unity app loads [↑](#footnote-ref-21)
22. <https://github.com/Unity-Technologies/UnityCsReference/tree/2019.4> [↑](#footnote-ref-22)
23. 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] [↑](#footnote-ref-23)
24. 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. [↑](#footnote-ref-24)
25. **I**nternational **O**rganization for **S**tandardization [↑](#footnote-ref-25)
26. **E**uropean **C**omputer **M**anufacturers **A**ssociation is a standards organization for IT related fields. [↑](#footnote-ref-26)
27. **M**icrosoft **I**ntermediate **L**anguage [↑](#footnote-ref-27)
28. **F**rames **p**er **s**econd. In Unity fps can be calculated by using the Update message. [↑](#footnote-ref-28)
29. Heap is the portion of the memory where the dynamically allocated memory resides. [↑](#footnote-ref-29)
30. 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. [↑](#footnote-ref-30)
31. Stack is an array of memory in a LIFO (Last In First Out) structure. [↑](#footnote-ref-31)
32. Boxing is when the CLR wraps a value type inside a System.Object instance and stores it on the heap. [↑](#footnote-ref-32)
33. int, long, float, double, bool, char, enum etc. [↑](#footnote-ref-33)
34. 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. [↑](#footnote-ref-34)
35. 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. [↑](#footnote-ref-35)
36. An extremely loose license with only minor restrictions. [↑](#footnote-ref-36)