Általános információk

A diplomaterv szerkezete:

1. Diplomaterv feladatkiírás
2. Címoldal
3. Tartalomjegyzék
4. A diplomatervező nyilatkozata az önálló munkáról és az elektronikus adatok kezeléséről
5. Tartalmi összefoglaló magyarul és angolul
6. Bevezetés: a feladat értelmezése, a tervezés célja, a feladat indokoltsága, a diplomaterv felépítésének rövid összefoglalása
7. A feladatkiírás pontosítása és részletes elemzése
8. Előzmények (irodalomkutatás, hasonló alkotások), az ezekből levonható következtetések
9. A tervezés részletes leírása, a döntési lehetőségek értékelése és a választott megoldások indoklása
10. A megtervezett műszaki alkotás értékelése, kritikai elemzése, továbbfejlesztési lehetőségek
11. Esetleges köszönetnyilvánítások
12. Részletesés pontos irodalomjegyzék
13. Függelék(ek)

Felhasználható a következő oldaltól kezdődő Diplomaterv sablon dokumentum tartalma. Ügyeljen a konzulens nevét és a beadás évét jelölő szövegdobozokra, mert azokra külön ki kell adni a frissítést. A mezők tartalma a sablonban a dokumentum adatlapja alapján automatikusan kerül kitöltésre.

A diplomaterv szabványos méretű A4-es lapokra kerüljön. Az oldalak tükörmargóval készüljenek (mindenhol 2.5cm, baloldalon 1cm-es kötéssel). Az alapértelmezett betűkészlet a 12 pontos Times New Roman, másfeles sorközzel.

Minden oldalon - az első négy szerkezeti elem kivételével - szerepelnie kell az oldalszámnak.

A fejezeteket decimális beosztással kell ellátni. Az ábrákat a megfelelő helyre be kell illeszteni, fejezetenként decimális számmal és kifejező címmel kell ellátni. A fejezeteket decimális aláosztással számozzuk, maximálisan 3 aláosztás mélységben (pl. 2.3.4.1.). Az ábrákat, táblázatokat és képleteket célszerű fejezetenként külön számozni (pl. 2.4. ábra, 4.2 táblázat vagy képletnél (3.2)). A fejezetcímeket igazítsuk balra, a normál szövegnél viszont használjunk sorkiegyenlítést. Az ábrákat, táblázatokat és a hozzájuk tartozó címet igazítsuk középre. A cím a jelölt rész alatt helyezkedjen el.

A képeket lehetőleg rajzoló programmal készítsék el, az egyenleteket egyenlet-szerkesztő segítségével írják le.

Az irodalomjegyzék szövegközi hivatkozása történhet a Harvard-rendszerben (a szerző és az évszám megadásával) vagy sorszámozva. A teljes lista névsor szerinti sorrendben a szöveg végén szerepeljen (sorszámozott irodalmi hivatkozások esetén hivatkozási sorrendben). A szakirodalmi források címeit azonban mindig az eredeti nyelven kell megadni, esetleg zárójelben a fordítással. A listában szereplő valamennyi publikációra hivatkozni kell a szövegben. Minden publikáció a szerzők után a következő adatok szerepelnek: folyóirat cikkeknél a pontos cím, a folyóirat címe, évfolyam, szám, oldalszám tól-ig. A folyóirat címeket csak akkor rövidítsük, ha azok nagyon közismertek vagy nagyon hosszúak. Internet hivatkozások megadásakor fontos, hogy az elérési út előtt megadjuk az oldal tulajdonosát és tartalmát (mivel a link egy idő után akár elérhetetlenné is válhat), valamint az elérés időpontját.

Fontos:

* a szakdolgozat készítő/diplomatervező nyilatkozata (a jelen sablonban szereplő szövegtartalommal) kötelező előírás Karunkon, ennek hiányában a szakdolgozat/diplomaterv nem bírálható és nem védhető!
* mind a dolgozat, mind a melléklet maximálisan 15 MB méretű lehet!

Jó munkát, sikeres szakdolgozat készítést ill. diplomatervezést kívánunk!

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 6](#_Toc332798843)

[Abstract 7](#_Toc332798844)

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

[1.1 Formázási tudnivalók 14](#_Toc332798846)

[1.1.1 Címsorok 14](#_Toc332798847)

[1.1.2 Képek 14](#_Toc332798848)

[1.1.3 Kódrészletek 15](#_Toc332798849)

[1.1.4 Irodalomjegyzék 15](#_Toc332798850)

[2 Utolsó simítások 16](#_Toc332798851)

[Irodalomjegyzék 17](#_Toc332798852)

[Függelék 18](#_Toc332798853)

Hallgatói nyilatkozat

Alulírott **Rezeda Kázmér**, 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. 07.

...…………………………………………….

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

# 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 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

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 mechanic 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, Preproduction, Production, Postproduction.



Figure 1: 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 2: The cover of The First Tree.

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 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 [[1]](#footnote-1)

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[[2]](#footnote-2), advertises and small content DLCs[[3]](#footnote-3).

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 optimal path finding solutions for our use cases
* Setup performance tests for AI and physics common use cases in order to later ensure 60 frames per seconds on average
* Compare the performance of the available physics engines, in order to define which executes faster overall in our use cases.

Unity for example by default has 4 physics engines

* The built-in 3D physics based on an older Nvidia PhysX[[4]](#footnote-4) version
* The built-in 2D physics based on Box2D[[5]](#footnote-5) engine
* Unity physics, a complete deterministic rigid body dynamics and spatial query system written entirely in high performance C# using DOTS[[6]](#footnote-6)
* Havok Physics, an implementation of the Havok physics[[7]](#footnote-7) 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[[8]](#footnote-8) 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 3: 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 the 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 for your 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 you 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 to test the game’s performance in the large. On these collected information later, usually in the post-production phase after some optimization 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 4: A seven-phase development pipeline

## Formázási tudnivalók

A dokumentum folyószövegéhez használjuk a **Normál** (angol Word esetén Normal) stílust.

### Címsorok

A fejezetcímek esetén a **Címsor 1-4** (Heading 1-4) stílusokat használjuk.

### Képek

A képhez használjuk a **Kép** stílust.

Képaláírást a képen jobb gombbal kattintva a Képaláírás beszúrása… opcióval adhatunk hozzá, így az automatikusan **Képaláírás** (Caption) stílusú lesz.



2.1. ábra: Példa képaláírásra

### Kódrészletek

Kódrészletek beillesztése esetén használjuk a **Kód** stílust.

using System;

namespace MyApp

{

class Program

{

static void Main( string[] args )

{

Console.WriteLine( "Szia Világ!" );

}

}

}

### Irodalomjegyzék

Az Irodalomjegyzékben szereplő hivatkozásokat **Irodalomjegyzék sor** stílussal formázzuk, a címüket pedig **Irodalomjegyzék forrás** stílussal emeljük ki.

A szövegbe a hivatkozásokat a Kereszthivatkozás beszúrása (Insert cross-reference) funkcióval helyezzük el (példa egy így beszúrt hivatkozásra: [1]), így azok automatikusan frissülnek a hivatkozások átrendezésekor.

# 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.

Irodalomjegyzék

1. Levendovszky, J., Jereb, L., Elek, Zs., Vesztergombi, Gy.: Adaptive statistical algorithms in network reliability analysis, Performance Evaluation - Elsevier, Vol. 48, 2002, pp. 225-236
2. National Istruments: LabVIEW grafikus fejlesztői környezet leírása, <http://www.ni.com/> (2010. nov.)
3. Fowler, M.: UML Distilled, 3rd edition, ISBN 0-321-19368-7, Addison-Wesley, 2004
4. Wikipedia: Evaluation strategy, <http://en.wikipedia.org/wiki/Evaluation_strategy> (revision 18:11, 31 July 2012)

Függelék

Eddig innen vettem dolgokat

Bevezető: <https://en.wikipedia.org/wiki/Program_optimization>

Pipelinról cucco: https://www.cgspectrum.com/blog/game-development-process

1. 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-1)
2. **I**n-**a**pp **p**urchases is a common way to monetize free to play games [↑](#footnote-ref-2)
3. **D**own**l**oadable **c**ontent is an expansion to a game in order to extend it with more playable content. [↑](#footnote-ref-3)
4. 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-4)
5. Box2D is an open source 2D physics engine written by Erin Catto [↑](#footnote-ref-5)
6. **D**ata-**O**riented **T**echnology **S**tack, a solution by Unity to take advantage of modern multicore processors [↑](#footnote-ref-6)
7. Havok physics is an industry leading complete physics solution for games and modelling software [↑](#footnote-ref-7)
8. **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-8)