University of Europe for Applied Sciences

Bachelor Thesis

Performance Comparison: OOP versus DOTS Paradigms in Mobile Games Made with Unity Engine

How does Object-Oriented Programming (OOP) compare with the Data-Oriented Technology Stack (DOTS) paradigms impact game performance made in Unity Engine for mobile devices?

Student: Alan Berberov
Matriculation Number: 48831137
Institution: University of Europe for Applied Sciences
Department: Game Design
Semester: 6 (SoSe2024)
First Advisor: Prof. Stephan Günzel
Second Advisor: B.A. Tobias Heukäufer
E-mail: alan@berberov.dev

Abstract

The thesis paper explores the analysis of multiple programming paradigms and their effect on performance in Unity Engine for mobile devices, highlighting the most commonly used Object-Oriented approach and Data-Oriented Technology Stack (DOTS), which empowers game developers to efficiently expand processing capabilities while maintaining high performance. The paper investigates these two methods by defining their core principles, advantages, and drawbacks while evaluating how they can theoretically optimize mobile games made in Unity Engine. It covers the fundamental elements of mobile devices and possible optimizations for both OOP and DOTS. Additionally, the paper describes the technical aspects of paradigms along with the smartphones' hardware. Furthermore, it explores existing projects that have employed these paradigms, considering their success in improving mobile game performance to identify best practices. In the end, analyzing the current market of mobile games with DOTS and outlining concerns of a new architecture.

Table of Contents

List of Figures

1. Introduction

Each year, the game industry experiences growth alongside the increasing quality of games. While a significant number of the population worldwide lack access to high-performance hardware for gaming, developers optimize games to ensure an enjoyable experience across different platforms. Nowadays, the most popular gaming device in the world is a smartphone (Clement, 2024). Its popularity has been gained for various reasons, such as affordability and mobility, providing a larger audience access to games while also allowing gamers to play almost anywhere (Zulhusni, 2023). Knezovic (2024) highlights that the current biggest market in the world is Asia Pacific, which reached 84.1 billion US dollars in 2023, confirming the rise of the popularity of mobile gaming in third-world countries (United Nations, 2021).

On the other hand, the advantages of mobile devices result in lower performance, which limits the gaming possibilities. Moreover, the lack of a cooling system forces smartphone manufacturers to limit CPU and GPU usage to avoid the device damaging itself (Turpeinen, 2020; Stealey, 2022).

The most popular game engine, the Unity Engine (Toftedahl, 2019), created a new architecture that claims to "scale processing in a highly performant manner" by developing projects using a Data-Oriented paradigm. This system has been called Data-Oriented Technology Stack or DOTS, and it is compatible with mobile devices. This system allows to storage of data efficiently in memory, writing effortless multithreaded code, and much more (Unity, no date).

Considering these factors, this Bachelor's thesis aims to explore the potential of different programming paradigms in improving the gaming experience for mobile devices.

2. Overview & Terminologies

This chapter will cover numerous critical terminologies alongside a brief overview and history of terms relevant to this thesis.

2.1 Programming Paradigms

A programming paradigm is a way of writing code that conveys the intentions of a programmer (Bobrow, 1984). Linda Wieser Friedman (1991) says that "it is an approach to solving programming problems." Many people confuse programming paradigms with programming languages. Programming languages follow specific rules that cannot be violated when the programming paradigm is an abstract structure of code that can be changed depending on specific requirements. It would be wrong to say that these concepts are independent because programming paradigms are relatively heavily connected with programming languages, for instance, C# and Java being Object-Oriented focused languages. At the same time, C and Basic can be classified as Procedural Programming languages. It is important to mention that programming languages can support multiple paradigms if they meet basic requirements. Bobrow (1984) highlights that in order to consider it supported, a language must be able to run programs effectively using this style (paradigm), as quantitative changes in running time result in qualitative changes in a system. Programs written in a paradigm should be clear, so it must provide composition techniques, the paradigm's primitives, and an appropriate user language (Bobrow, 1984).

2.2 Object-Oriented Programming

One of the most popular and successful areas of computer science is Object-Oriented Programming or OOP (Van Roy and Haridi, 2004). Black (2013) says that over the 1960s Ole-Johan Dah has been working on the initial concepts of the OOP and has been creating a compiler for the programming language named SIMULA. This language was designed in 1963, published the first version, which is known as SIMULA I, in 1965, and was released in 1967 and was named accordingly SIMUAL 67 (Black, 2013). However, it wasn't until the early 1980s that OOP gained industrial popularity that C++ made its appearance (Stroustrup, 1997). Another significant development was Smalltalk-80, which was released in 1980 as the outcome of research conducted in the 1970s (Van Ray and Haridi, 2004). According to Stroustrup (1993) and Kay (1993), both of these languages were strongly impacted by SIMULA 67.

The foundation of Object-Oriented Programming is the concept of object. An object can be characterized as a data field with distinct attributes and behavior. Rather than emphasizing the logic required to manipulate objects, OOP focuses on the objects that developers want to manipulate. Large, complicated programs that are regularly updated or maintained fit well with this programming style, since it allows dividing projects into smaller independent groups, while also including code reusability, scalability, and efficiency (Gillis, 2021).

I find it is important to also explain the basic terminology of OOP based on Gillis (2021):

- Class is a predefined abstract blueprint containing data, methods, etc.
- Object is a concrete instance of a class with specifically defined data.
- Method is a function, which is defined inside of a class and represents its behavior.
- Attribute is a data field that is defined in a class template. It stores data of an object and describes the object's state.

2.3 Data-Oriented Technology Stack (DOTS)

The recently released Unity Data-Oriented Technology Stack (DOTS) is not a pure programming paradigm, but rather a complex architecture consisting of "a combination of technologies and packages that delivers a data-oriented design approach to building games in Unity" (Unity, no date). Unity (no date), highlights that DOTS is built around of main 3 parts that are ESC, Burst Compiler, and C# Jobs System. I will talk about the last two in section 4.1, while now will focus on ESC and Data-Oriented Programming. Although Data-Oriented design has existed in one form or another for decades, Noel Llopis was the first to formally name it in his September 2009 article (Fabian, 2018). In the same article Llopis (2009), mentions the main disadvantage of Object-Oriented Programming: by definition, OOP only operates one object. Meanwhile, the majority of the objects in a greater number of games never use objects only once. It is common to reuse code, but OOP treats every object independently of one another, ignoring that (Llopis, 2009).

Game data is naturally organized in the manner of inheritance trees, containment trees, or message-passing trees, which is what comes to mind when we think of objects. For this reason, every time an object is edited, it results in that object being able to access objects further down the tree. When a program loops through a group of objects performing an identical operation, each object goes through a cascade of completely different operations (Llopis, 2009).

Decomposing each object into its parts and grouping components of the same type in memory, regardless of the object they come from, can help achieve the best possible data layout. This organization creates large, homogeneous blocks of data that allow us to process the data sequentially (Llopis, 2009).

2.4 Mobile Devices

Smartphones have become the most popular gaming device worldwide (Clement, 2024), and a few practical distinctions still exist compared to personal computers (Turpeinen, 2020).

First of all, mobile devices are designed to be portable which makes them inconsistent in terms of electricity or the equivalent computational power. Resulting lack of demand for a cooling system such as a fan or a heat sink, so mobile devices tend to rely on a concept called thermal throttling (Turpeinen, 2020).

According to Stealey (2022), thermal throttling of a CPU is a process that artificially slows down its speed meaning less processing power, when the temperature of a CPU exceeds a certain threshold. Hence, it is cutting down the performance resulting in a lower frame rate but keeping the CPU from overheating and potentially damaging itself.

In 2016 Prakash et al. talked about modern smartphones being built on the architecture called MultiProcessor System-On-Chip (MPSoC), which is a variation of SoC with multiple CPU cores and it is important to notice that most of the System-On-Chip are MPSoC (PCMAC, no date). The SoC or System-On-Chip is a chip that includes various components, such as CPU, GPU, memory, etc., in one chip alongside software resulting in better performance, less power usage, and making it more compact (Intel, 2024).

The CPU and GPU are commonly the major processing elements in computers, while in contrast on mobile devices, to maintain all active apps and keep the ability to switch between them the CPU is put under high pressure, while also its frequency is ruled by the operating system. Meanwhile, the GPU is operating more graphics-related specific tasks and
device drivers individually adapting their frequency (Prakash et al., 2016; Stealey, 2022).

3. Overview of Object-Oriented Approach

3.1 Core Principles

Object-Oriented Programming is based on four core principles, also known as the four pillars of OOP:

Encapsulation
Encapsulation is the process by which an object's internal data is closed in a protective capsule from the use of external entities. In other words, this concept is based on hiding the inner representation from the outside of the definition and assembling a single unit (object) that contains essential data and methods to operate (Masumi, no date; Taylor, 2024).

Encapsulation allows the creation of access regulations, which helps to define the accessible and modifiable data for different types of objects. Moreover, it separates external interactions from the internal state of an object by stopping unintended interference which leads to a security increase. Encapsulating data and segregating public interfaces of an object from irrelevant information for external users about the actual implementation pieces allows developers to work in parallel since changes to an object's internal structure can occur without changing the code that interacts with it (Taylor, 2024).

The benefits of encapsulation are reducing system complexity, preventing users from setting the internal data of the component into an invalid or inconsistent state and helping to make Unit tests (GfG, 2024).

Inheritance
Inheritance is a principle that provides an opportunity to create a subclass, which inherits attributes and methods from an already existing class (Taylor, 2024). As a result, a new class can be called a child class, subclass, or derived class, while an existing class is called a parent class, base class, or superclass (Masumi, no date). This concept enhances code extensibility and reusability of existing objects.

Inheritance is different from language to language, for instance, GfG (2024) talks about 5 different types in Java, which comes down only to 2 main ones, since others are combinations of those two. Single inheritance is when class B can inherit data and functionality from class A; Multiple inheritance is the same but allows multiple parent classes, meaning class C can inherit data from A and B simultaneously.

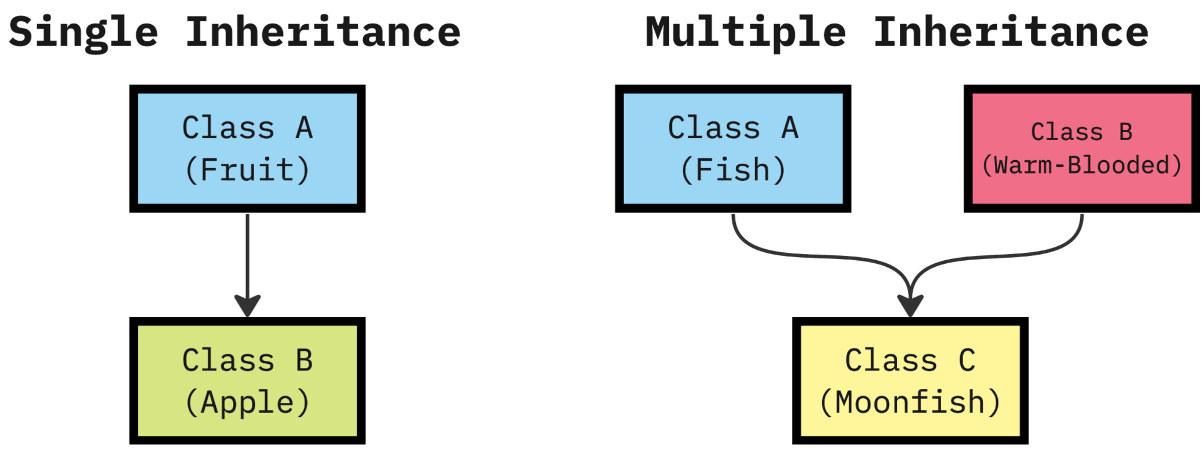


Figure 1: Figure: Inheritance Example

Abstraction

Abstraction is based on creating classes that are not focused on the concrete implementation, but on their interfaces instead (Masumi, no date). In other words, it is a manner of coding where complex systems are simplified and emphasize critical elements, while concealing unnecessary features (Taylor, 2024).
It is important to note that more abstraction usually means more coupling, which may cause issues if code needs to be changed (CodeAesthetic, 2022).

Polymorphism

Polymorphism allows multiple ways of performing the same method with different implementations. There are 2 types of polymorphism, Compile-time Polymorphism (Static) and Run-time Polymorphism (Dynamic) (Masumi, no date).

Compile-time Polymorphism can be accomplished by a concept named Method Overloading. Method overloading occurs when different functions in a class share the same name but have different signatures. As a result, overloaded methods accept different arguments, which may differ in terms of number or type (GfG, 2024).

When the compiler is unable to distinguish between a parent class method and a child class method it is called a Run-time Polymorphism. Method overriding is the core conception of Dynamic polymorphism, which occurs when a subclass method is overridden by a method from the superclass (GfG, 2024).

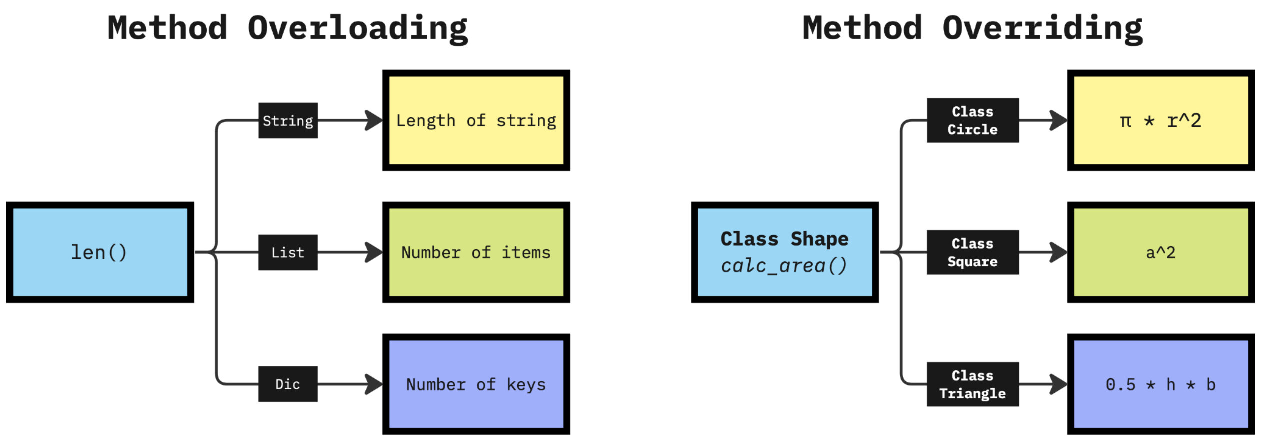


Figure 2: Figure: Example of polymorphism in Python

4. Overview of Data-Oriented Technology Stack

4.1 Core Principles

Data-Oriented Design is a relatively young paradigm created in 2009 compared to OOP which was created in the 70s and has not established clear principles such as OOP did (Black, 2013; Fabian, 2018). Nevertheless, the common elements can be traced throughout different research:

Data Decoupling
Fabian (2018) talks about Object-Oriented Programming creating data with an already predefined context - an object. However, this data reusability becomes limited, and further development creates complex objects with unrelated data and functions, which usually require additional data, mixed. Data-oriented design solves this by separating code from data and structuring this data more efficiently.

Optimizing Memory Access
This principle implies that all data should be designed to be organized in memory in an effective way for the CPU to access it. Architecting software data to align with the underlying memory subsystem, which will enable modern processors to operate at their maximum capacity, is a fundamental approach of DOD (Dang, 2023). A more detailed investigation of memory allocation will be covered in the section "4.4 DOTS Memory".

4.2 DOTS Structure

Entity Component System or ECS is an architecture that started to develop in the early 2000s in the game development industry and did have multiple names: Entity Systems and Component Systems. A noticeable investment was made by Scott Bilas at a GDC talk in 2002 (Martin, 2007). Nowadays, ECS in Unity is a pattern that separates code logic from data in an efficient manner (Smith, 2018).

Component

Components are fragments of game data, such as World Position (Vector3), Velocity (Vector3), Health (float), etc. Components can also be empty, which can provide the functionality of tags. For instance, an entity can be marked as broken if a "Broken" component is attached to it (Smith, 2018).

Systems

On the other end, all of the component data transformation is done by Systems. For instance, all moving entities' positions might be changed by their velocity by the time passed since the previous update with a corresponding system (Unity, 2020b).

Entities

Entities in their core are IDs that can be seen as individual "objects" in a game world. Instead of having behavior or data, it serves as a connector or a uniting element for data (Unity, 2020a). An entity is a group of indexes of different components' arrays in memory (Smith, 2018). This concept is known as AoS or Array of Structs and is described in more detail in the "4.4 DOTS Memory" section.

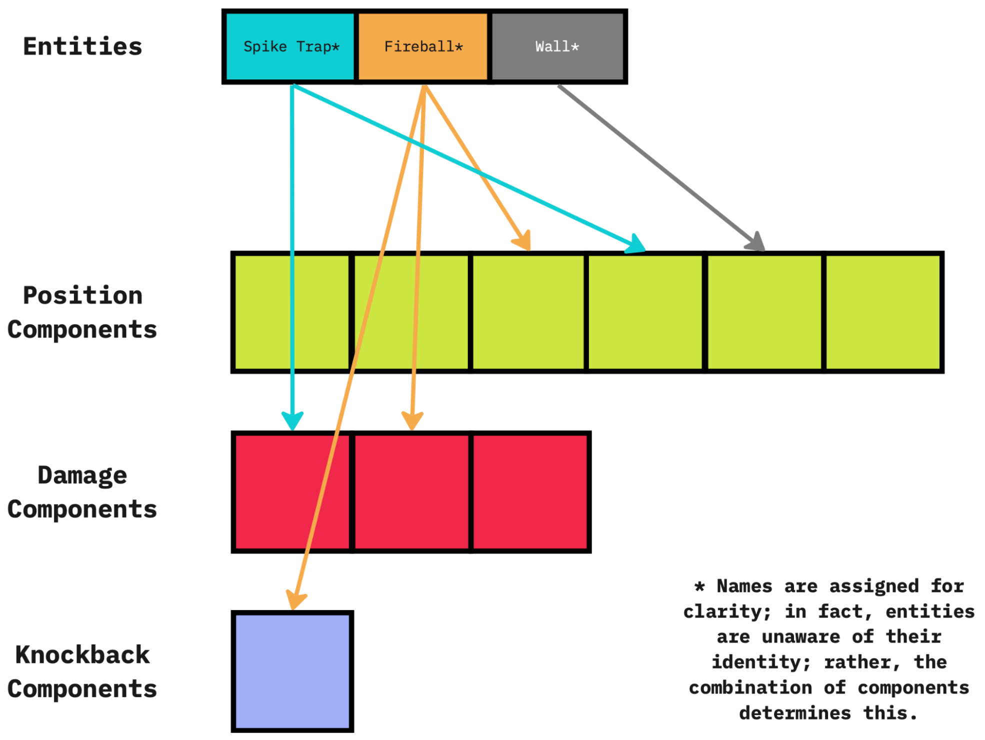

Entities
Spike Trap*
Fireball*
Wall*
Position Components
Damage Components
Knockback Components
* Names are assigned for clarity; in fact, entities are unaware of their identity; rather, the combination of components determines this.

Figure 3: Figure: Entities model example

As mentioned earlier, Unity DOTS is based on three main packages: ESC, Burst Compiler, and C# Job System (Unity, no date). The last one enables users to create multithreaded code that works seamlessly with Unity Engine (Unity, 2024). Multithreading is a programming technique that allows a CPU to operate many threads concurrently across its multiple cores. Tasks or instructions are executed simultaneously rather than sequentially (Unity, 2020c).

The Burst compiler, when combined with Unity's Job System, optimizes C# code by converting the usual IL/.NET bytecode to efficient native CPU code using the LLVM compiler, resulting in increased performance (Unity, 2023a).

4.4 DOTS Memory

Smith (2018), in a Unite LA talk, outlines that CPU speed has grown drastically during the last couple of decades at a point where single-core processor evolution slowed down noticeably, which led to the manufacturing of multiple-core processors. On the other side, the speed of DRAM (Dynamic random-access memory) has not increased as nearly closed. As a result, processing data became extremely cheap in a performance sense, while accessing this data turned into a performance bottleneck.

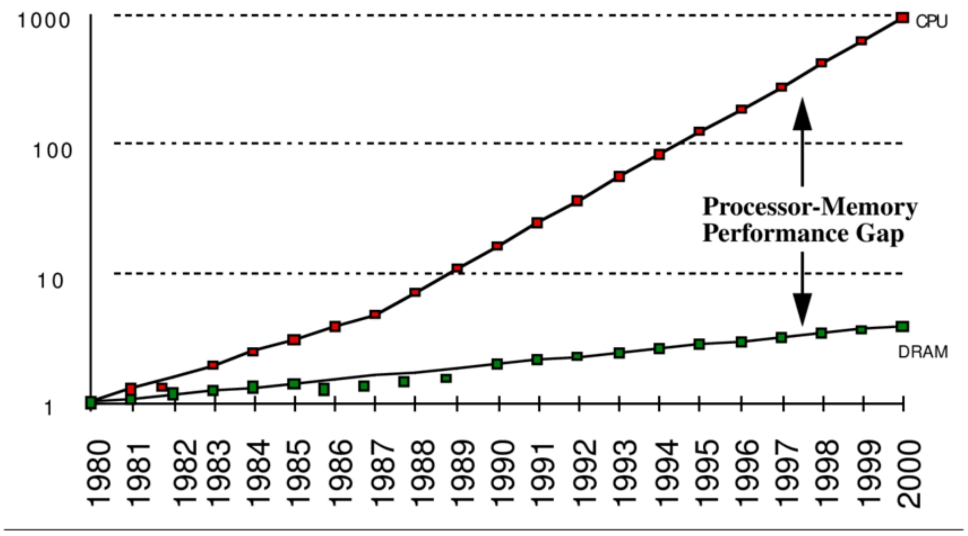


Figure 4: Processor-Memory Performance Gap (Patterson et al., 1997)

One of the possible solutions to this is to structure data better in a way the CPU could find it faster in memory (Smith, 2018). If the incorrect data is loaded into the cache because it is inefficiently arranged in the memory, then the time it takes to read the correct data is considerably longer than if the data could be prefetched. The extra time cost for accessing data farther from the processor also becomes an extra power cost because transferring data costs more energy than processing that data (Bayliss, 2022). For instance, there is a comparison of how OOP data is structured versus in DOD (Smith, 2018):

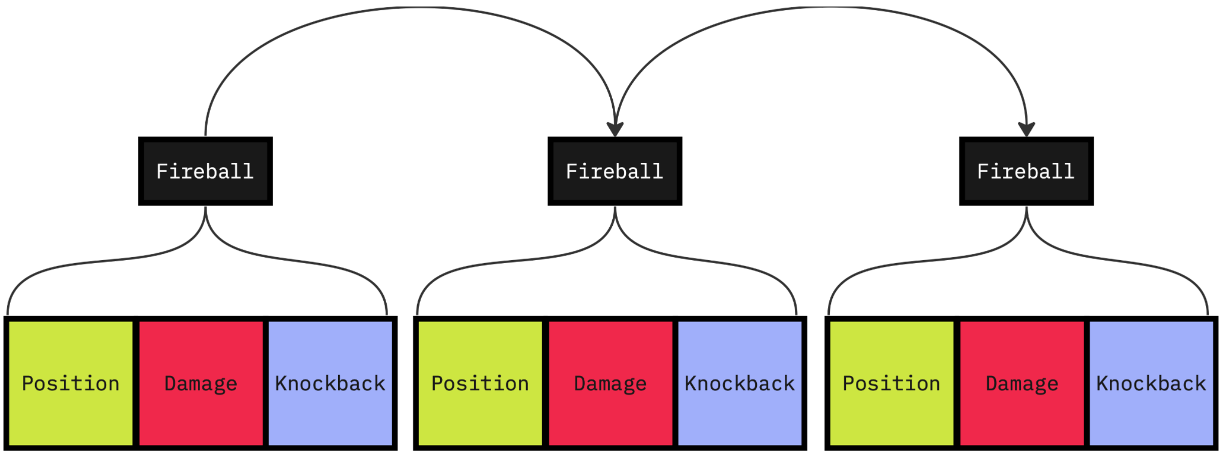


Figure 5: Structure of Arrays

The diagram above shows the structure of the data in OOP, where each fireball object allocates RAM individually containing all data associated with this object. This layout is called SoA or Structure of Arrays, which may be inefficient for the reasons provided above.

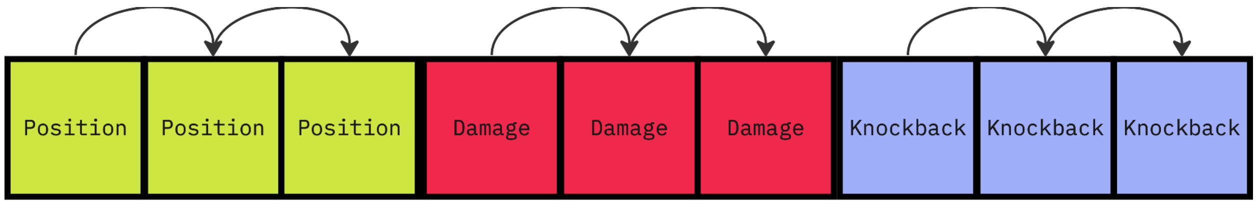


Figure 6: Array of Structs

On the contrary, this table shows how the DOD approaches data allocation. First and foremost, it is noticeable that all data lives independently from the object, meaning iteration through each data would be faster since every data for a specific system is lined up in a fast-access way for the CPU. This allows us to avoid iterating through each NPC individually, but to loop through all similar data, such as processing all movement at once, then all damage at once, etc. This way of organizing data is called AoS or Array of Structs.

This separation of data makes it easier to execute code in parallel. This indicates the straightforward option to use multithreading, which requires a significant effort in Object-Oriented programming (Smith, 2018).

5. Comparative Analysis

5.1 3D Crowd Rendering

In 2020, a master's student in Computer Science and Engineering, Max Turpeinen, conducted research on 3D Crowd Simulation on mobile devices using Unity's MonoBehaviours as an Object-Oriented Approach and Unity DOTS and compared performance with a different simulation scarious.

Shopf et al. (2008) outline that in massive crowd simulations, the bottleneck of animations is becoming a CPU and not memory. The reasoning for that is a CPU needs to process all required calculations for blending various animations, which highlights the individualism of all characters. On top of that offsetting their animation randomly to avoid synchronization adds an extra layer of complexity for a CPU.

For the performance tests, 2 different Unity projects were created on Unity Version 2019.3.0f3 Personal. During the time period of this research, the DOTS architecture was actively in development, and numerous packages were only available as preview versions. The important packages for the DOTS, such as Burst and Entities were at versions 1.2.1 and 0.5.1 respectively (Turpeinen, 2020).

Methodology and Implementation

The hardware to run these simulations was an iPhone 6S and an iPhone XR with 32 GB and 64 GB respectively, while both of the devices had the same iOS version 13.4.1. In order to build applications for this platform Unity-compatible software (Xcode) was used on version 11.4.1 (Turpeinen, 2020).

The render pipeline was changed by lowering the rendering scale for both of the devices to 0.81 for the iPhone 6S and 0.665 for the iPhone XR, resulting in a fair comparison since the screen resolution of the devices differs, 1334 x 750 and 1792 x 828 respectively. The development of both implementations was been done in parallel to avoid any variation except the programming paradigm for fairer results (Turpeinen, 2020).

The scenarios were aimed at looking at the problem from different perspectives by changing the number of agents with assumptions that DOTS show a better performance, while also increasing the variety of agents and animation which will give an advantage for the OOP approach assummably (Turpeinen, 2020).

CPU, GPU usage, and FPS (frames per second) were utilized performance as benchmarks which were gathered with Xcode. The frame rate has been limited to 30 frames per second, meaning if the CPU or GPU exceeds 33ms it will be considered as overworked (Turpeinen, 2020).

Other minor optimizations have been done to minimize the difference between the two, but it is out of scope for this section.

First Scenario

The y-axis represents 2 units at the same time: fps on the left for the FPS bar and ms on the right for the GPU and CPU. The x-axis illustrates the number of characters animated simultaneously, and OO is the OOP approach using GameObjects and MonoBehaviour (Turpeinen, 2020).

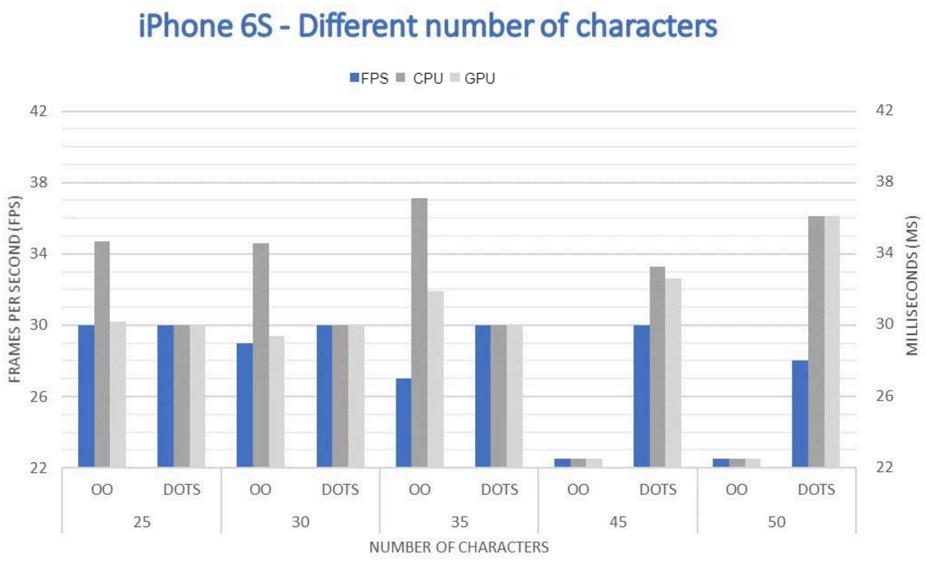


Figure 7: Figure: iPhone 6S - Different number of characters (Turpeinen, 2020)

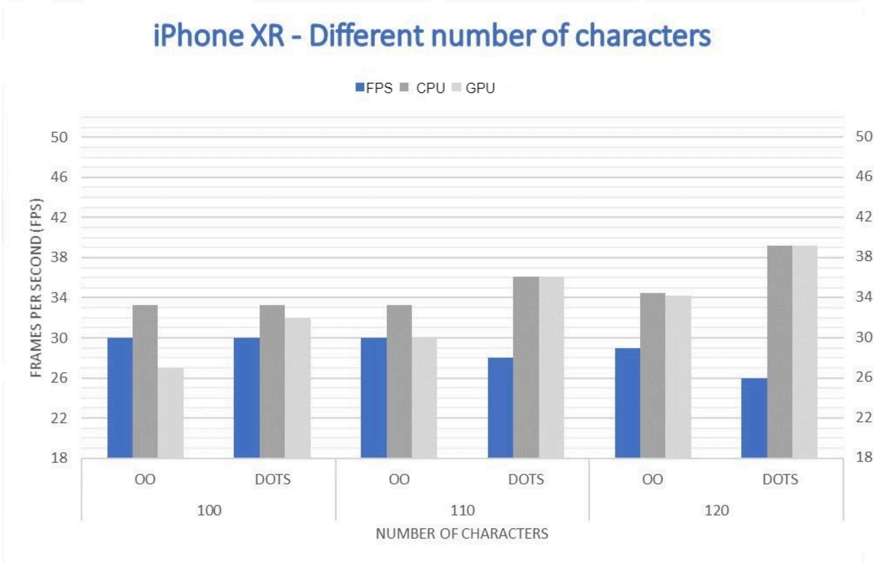


Figure 8: Figure: iPhone XR - Different number of characters (Turpeinen, 2020)

The first comparison scenario was operating one type of character with only one animation. iPhone 6S's simulation chart clearly shows the advantage of DOTS, while on the iPhone XR, both of the paradigms showed similar robust results with a tiny lead to

OOP. Analyzing this outcome and the lack of information about code structure, the reasons for a slightly worse performance of DOTS may be the following:

As mentioned earlier, in massive crowd simulations bottleneck is processing part of the CPU and not memory accessing (Shopf et al., 2008). Hence, the optimization of data in memory in ECS does not contribute to the result so heavily, still then logically multithreading should improve the performance regardless since data will be computed in parallel. However, jobs are meant to be small units that perform a specific task (Unity, 2024), but if the size of a data package grows the risks of Job System overhead are higher, which may result in a higher waiting time (Vacheresse, 2023).

Adding to that, based on Unity (2023b) documentation, managed components cannot be used in jobs, or Burst compiled. Managed components are components that can store any properties including reference types, unlike unmanaged components, which are pure struct value types. Another important difference is their need for garbage collection and the way of storage. Unmanaged components are stored directly in chunks, what exactly is a chunk is not relevant in this context, but it is a group of components that is organized in an efficient way for memory, while managed components are stored in one array for the whole project, and chunks stores indices of these components (Unity, 2023b). This information is relevant because animations with ESC do not have a traditional solution, instead, there are numerous ways to implement it, such as (Iclemens, 2023):

- **The hybrid approach.** Combination of standard GameObject animation tools with ESC architecture, which can be achieved by synchronizing entities with GOs.
- **Pure ECS Animation.** This method is based on animating meshes using components and systems only.
- **Vertex position baking.** This technique can be achieved by baking vertex positions into a texture and send to GPU, where a specific shader would animate corresponding meshes. Great talk by Jérémy Guéry (2023) about this topic on "Unity at devcom with IXION".
- **Etc.**

It is important to understand that this research does not provide a concrete technique that has been used, so it would be fair to assume that it has been done with the hybrid approach since it requires the least amount of time to implement, while also providing a familiar interface using build-in MonoBehaviour based Animator component (Iclemens,

2023), and that the author would cover a custom technique in the opposite case is also reasonably presumed.
WYAN Games (2022), describes the creation of an animation system with a Hybrid approach and uses managed components to achieve synchronization between ESC entities and GameObejcts with Animator scripts attached to them.

As a result, this simulation might not be using all the power of DOTS and completely misses the performance increase from the Jobs System and Burst Compiler (2023 MANAGED COMPONENTS).

Second Scenario

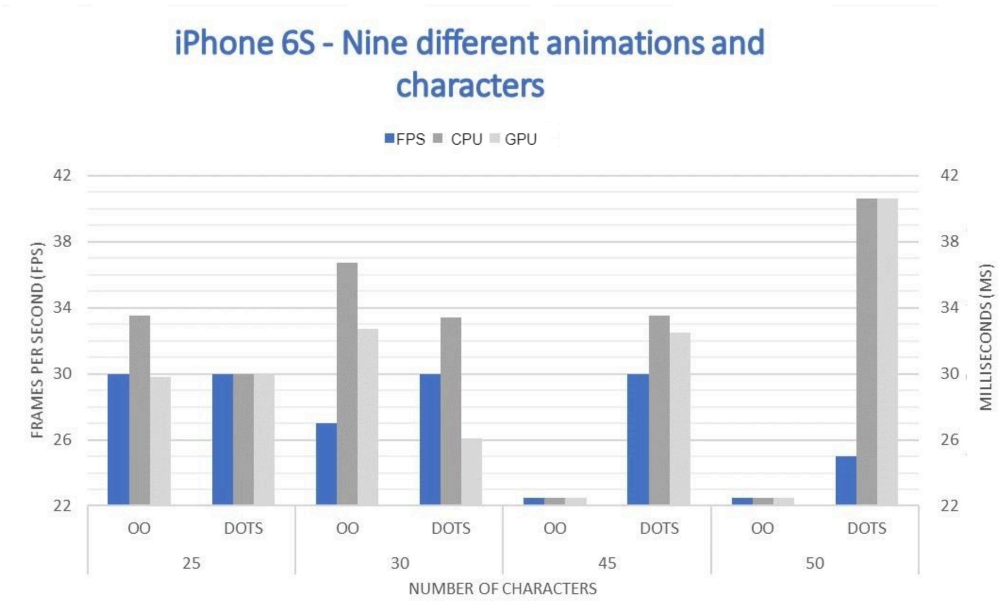


Figure 9: Figure: iPhone XR - Nine different animations and characters (Turpeinen, 2020)

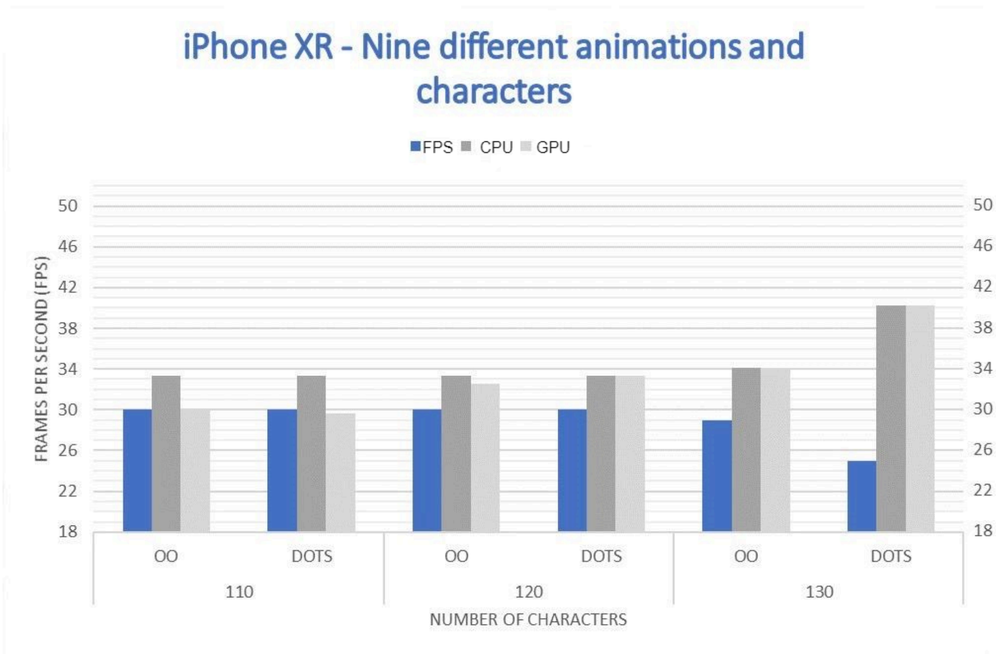


Figure 10: Figure: iPhone XR - Nine different animations and characters (Turpeinen, 2020)

The second setting operated multiple, to be exact nine, character types, with a unique animation each (Turpeinen, 2020). On the iPhone 6S, even 25 agents caused the CPU to overwork by surpassing the 33ms mark. The further increment of agents caused an FPS drop until dropped and presumably a thermal throttling has been activated (section 2.4). DOTS in this simulation had been showing a consistent result with 30 fps until the 50-agent mark, but CPU time exceeded 33ms.

However, on iPhone XR, DOTS architecture performed worse than OOP. Surprisingly, OOP has accomplished even more consistent results than with one type of character. Multiple reasons could be provided there, first of all, Turpeinen (2020) talks about how each test had to be conducted with phones connected to the PC with a charging cable for testing purposes, which may affect the battery temperature and cause thermal throttling from session to session. This might have affected performance insignificantly, but it would be fair to notice that fluctuations in performance are relatively small as well. Secondly, the OOP approach goes through each object individually, and having multiple types of characters with different animations should not influence its performance drastically, which is the opposite for DOTS, since it manages same-type components simultaneously, meaning a decrease in performance with a rising number of variants.

Third and Fourth Scenarios

In order to conduct a simulation with larger agent quantities, the mesh size has been reduced firstly to 20% of the original mesh and then to 10% in a fourth scenario (Turpeinen, 2020). A similar approach, which reduces a geometrical complexity, for instance, decreasing the number of polygons in a mesh, resulting in a higher performance, is commonly used in game development and is called Level of Detail or LOD (Wißner, Kistler and André, 2010).

For the iPhone 6S, results were the same, but for the iPhone XR, the decrease of mesh to 20% resulted in a DOTS architecture taking over and showed consistent 30fps without CPU or GPU overwork at 600 characters, while OOP started to lag at 500 and fell drastically at 540 agents.

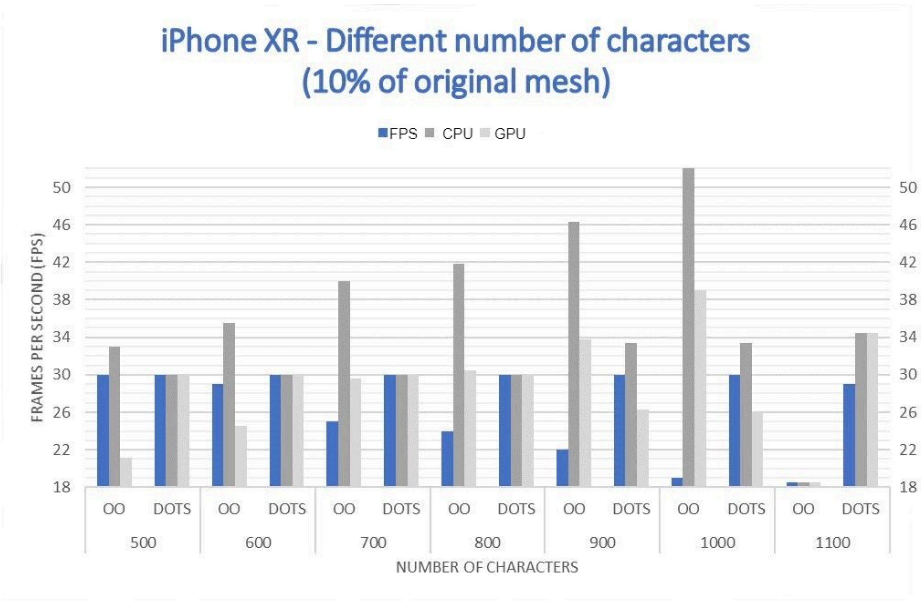


Figure 11: Figure: iPhone XR - Different number of characters (10% of original mesh) (Turpeinen, 2020)

However, by lowering the mesh resolution even further up to 10% of the original, the results on the iPhone XR became even more distinct. As illustrated in the chart above, at 500 units, the OOP system almost broke through the mark of 33ms on the CPU, while the GPU was relatively inactive. At 1000 agents, the CPU's processing power has been reaching its limits, and at 1100, it faces thermal throttling. On the other side, the DOTS design managed to equally use GPU's and CPU's power, resulting in a consistent 30fps and absence of major processing power spikes. At only 1100 characters, fps dropped by one unit.

Summarizing the aforesaid points, it is clear that in a 3D crowd simulation DOTS performs better with higher quantities of agent scenarios. It is also more reliable on older devices such as the iPhone 6S. However, when dealing with smaller crowds with higher computational power required, it seems to lose to OOP. If my assumptions are correct, it is caused by a lack of Burst Compiler and Jobs System support, which leads to less optimized processing performance. So, when the agent count is in a relatively small range of 100-150 agents, the bottleneck is the CPU's processing power. However, as the number of agents increases, it shifts towards memory access speed, which DOTS undoubtedly has an advantage. Moreover, it is important to mention that the more variations are added, the more advantages OOP gains.

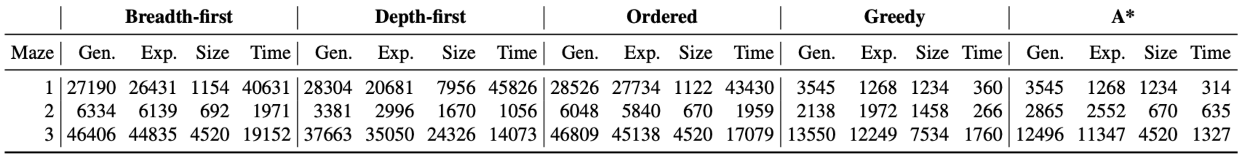
Turpeinen's (2020) research misses a part about the implementation, which leads to building assumptions about used techniques and underlying rocks but gives a decent comparison of various scales of crowd simulations on mobile devices and helps to outline the advantages and disadvantages of both approaches. Unity DOTS is a relatively new system lacking research on this topic, which may only assume how different animation approaches can enhance DOTS even further, for instance, by using Burst code and Jobs (Iclemens, 2023).

5.2 Artificial Intelligence

Sunblink (2022), a game studio, released a "Hero-based castle-defense" game named HEROish, which was developed using DOTS with the help of the Apple Arcade team for various platforms, including mobile (Larrabee, 2023).

Based on Larrabee's (2023) talk, during the beginning of the development stage, the minimal requirements were stable 30fps on a single-core 2015 tablet on the low end. Furthermore, having as many units as technically achievable was one of the early design pillars. The technical team outlined that AI, pathfinding, steering, and unit avoidance would be performance-critical points. The gameplay is supposed to have units' formations and dynamically changing priority points for pathfinding, which would consume even more of the CPU power. During the testing of a unit avoidance technique, one update call consumed 20ms on average on a desktop after converting the algorithm into a Data-Oriented approach with unmanaged structs with native data types and Jobs, resulting in each update consuming only 4.5ms while adding a Burst compiler, it reduced time even further to 0.4ms. It is important to mention that the OOP code had the potential for optimization, but as Justin Larrabee said, converting it to basic DOTS helped to avoid the "performance optimization nightmare."

According to Silva and Viella (2017), the pathfinding algorithm for mobile devices is performant-heavy, which often becomes a limitation of the gameplay. The following table provides information about different pathing techniques permanence in a 2D game on an Android phone Sony E1 Dual (512 MB RAM) and 1.2 GHz Dual-Core processor. However, this test was conducted in the Java language and did not use Unity as a game engine. Nevertheless, Java is an OOP language and is the native language of Android Studio, which presumably makes it legit to compare as a Unity-based game performance-wise. The table shows the result of different AI agents completing three mazes. The A* and Greddy algorithms outperformed others but still showed a relatively large time consumption of the calculations. For instance, the first maze did not contain any obstacles and had a size of 30x40 units, in this test A* illustrated the best time which was the 314ms. The second maze had walls and a size of 20x30, which increased pathfinding time to 635ms.



	Breadth-first				**Depth-first**				**Ordered**				**Greedy**				**A***			
Maze	Gen.	Exp.	Size	Time	Gen.	Exp.	Size	Time	Gen.	Exp.	Size	Time	Gen.	Exp.	Size	Time	Gen.	Exp.	Size	Time
1	27190	26431	1154	40631	28304	20681	7956	45826	28526	27734	1122	43430	3545	1268	1234	360	3545	1268	1234	314
2	6334	6139	692	1971	3381	2996	1670	1056	6048	5840	670	1959	2138	1972	1458	266	2865	2552	670	635
3	46406	44835	4520	19152	37663	35050	24326	14073	46809	45138	4520	17079	13550	12249	7534	1760	12496	11347	4520	1327

Figure 12: Figure: Execution data for each maze: Number of generated (Gen.) and expanded (Exp.) states, path size (Size), and runtime (Time) in milliseconds for each pathfinding algorithm (Silva and Viella, 2016)

In a different Unity project on PC, a primitive unoptimized A* pathfinding algorithm resulted in around 700ms for an AI to find a solution for a maze 5 times. However, after recreating the same algorithm implementing ESC, Jobs, and Burst, the result declined to 0.07ms on average (CodeMonkey, 2020). This example does not prove that the performance boost would be the same on mobile devices, for reasons such as mobile devices do not usually have as many CPU cores as PCs, which would result in less processing power from using multithreading. However, it does provide a strong trend of decreasing processing time for pathfinding, which presumably would also perform more efficiently on mobile devices and Larrabee (2023) confirms this but does not provide any concrete values.

5.3 Analyzing the Research Gap: DOTS on Mobile Devices

Unfortunately, there is not enough information about the performance of DOTS on mobiles in open access. The only resources accessible were talks and not advanced research. The reason for this is the lack of games created with Data-Oriented Technology Stack on mobiles. Hence, DOTS has a much steeper learning curve, which would force game studios to invest their time and money into developers adapting to a whole new paradigm, rather than investing in the development itself. DOTS is a relatively new architecture, and it is still in active development. Therefore, DOTS in the mobile devices field is not researched.

The objective of my thesis project is to investigate unstudied DOTS implementations for mobile devices, such as pathfinding, CPU-heavy rendering scenarios, and various AI behaviors.

6. Conclusion

Summarizing this research, Unity DOTS performs better in most of the scenarios. However, each case is unique, and in some of them, OOP can outperform DOTS, for instance, when the diversity of small quantities of objects is increased, as can be observed in a conducted test of animating 9 characters in section 5.1.

Optimizing the processing powers of the CPU and GPU on mobile devices is key to increasing performance and battery life since unoptimized usage would result in thermal throttling, which drastically lowers performance (Stealey, 2022). For instance, during crowd rendering, the CPU and GPU have been loaded equally on the instance using DOTS. This difference can help to avoid overheating of one particular unit by separating logic equally (Turpeinen, 2020). It minimizes the risk of thermal throttling and optimizes battery time (Stealey, 2022). Moreover, the Data-Oriented approach stores data in a format that lowers accessing time, which is usually a bottleneck in CPU operations (Smith, 2018), resulting in more performant CPU usage.

The thesis question of which paradigm performs more efficiently on mobile devices cannot be answered sufficiently because of the lack of research. Even though there are numerous reliable academic talks and research about DOTS and OOP, there is no legitimate research regarding the implementation of DOTS on mobile devices specifically and no direct comparison of DOTS and OOP performance impact on mobile devices. Moreover, only a few mobile games are made with DOTS, which only narrows the area to research. However, observing the given scenarios, it can be claimed that Unity DOTS is usually a more performance-efficient architecture for mobile games for several reasons listed above. However, there is a possibility that in some scenarios, Unity DOTS will perform worse than OOP.

The objective of my thesis project is to investigate unstudied DOTS implementations for mobile devices, such as pathfinding, CPU-heavy rendering scenarios, and various AI behaviors.

Bibliography

Bartoníček, J. (2014). Programming Language Paradigms & The Main Principles of Object-Oriented Programming. *CRIS - Bulletin of the Centre for Research and Interdisciplinary Study*, 2014(1), pp.93–99. doi:https://doi.org/10.2478/cris-2014-0006.

Bayliss, J.D. (2022). The Data-Oriented Design Process for Game Development. *Computer*, 55(5), pp.31–38. doi:https://doi.org/10.1109/mc.2022.3155108.

Black, A.P. (2013). Object-oriented programming: Some history, and challenges for the next fifty years. *Information and Computation*, [online] 231, pp.3–20. doi:https://doi.org/10.1016/j.ic.2013.08.002.

Bobrow, D.G. (1984). If Prolog Is the Answer, What Is the Question? *Fifth Generation Computer Systems*, pp.138–145.

Clement, J. (2024). *Top devices for gaming 2021*. [online] Statista. Available at: https://www.statista.com/statistics/533047/leading-devices-play-games/.

CodeMonkey (2020). *Pathfinding in Unity DOTS! (Insane Speed!!!)*. [online] www.youtube.com. Available at: https://www.youtube.com/watch?v=1bO1FdEThnU [Accessed 11 Apr. 2024].

Dang, T. (2023). *Revolutionize Your Code: The Magic of Data-oriented Design (DOD) Programming*. [online] www.orientsoftware.com. Available at: https://www.orientsoftware.com/blog/dod-programming/#:~:text=At%20its%20core%2C%20DOD%20prioritizes [Accessed 9 Apr. 2024].

Eliza Taylor (2024). *4 Principles of Object Oriented Programming: Explained in Detail*. [online] www.theknowledgeacademy.com. Available at: https://www.theknowledgeacademy.com/blog/principles-of-object-oriented-programming/.

Fabian, R. (2018). *Data-oriented design : software engineering for limited resources and short schedules*. Richard Fabian, p.3.

GeeksforGeeks. (2024). *Four Main Object Oriented Programming Concepts of Java*. [online] Available at: https://www.geeksforgeeks.org/four-main-object-oriented-programming-concepts-of-java/.

Gillis, A.S. (2021). *What is Object-Oriented Programming (OOP)?* [online] App Architecture. Available at: https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP.

Guéry, J. (2023). *Unity at devcom with IXION || Unity*. [online] www.youtube.com. Available at: https://www.youtube.com/watch?v=SjNE8BplBEk [Accessed 9 Apr. 2024].

Iclemens (2023). *Other - DOTS Animation Options Wiki*. [online] Unity Forum. Available at: https://forum.unity.com/threads/dots-animation-options-wiki.1339196/ [Accessed 9 Apr. 2024].

Intel (2024). *Chipset and System-on-a-Chip (SoC) Reference for Intel® NUC*. [online] Intel. Available at: https://www.intel.com/content/www/us/en/support/articles/000056236/intel-nuc.html.

Kay, A.C. (1993). The Early History of Smalltalk. *ACM SIGPLAN Notices*, 28(3), pp.69–95. doi:https://doi.org/10.1145/155360.155364.

Knezovic, A. (2019). *Mobile Gaming Statistics: 85+ Statistics for 2020*. [online] Udonis. Available at: https://www.blog.udonis.co/mobile-marketing/mobile-games/mobile-gaming-statistics.

Linda Weiser Friedman (1991). *Comparative Programming Languages*. Englewood Cliffs, N.J. : Prentice Hall.

Llopis, N. (2009). Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP). *Game Developer*, 16(8), pp.43–45.

Martin, A. (2007). *Entity Systems are the future of MMOG development – Part 1 – T-machine.org*. [online] t-machine.org. Available at: https://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/.

Masumi, S. (no date). *The four principles of OOP*. [online] Sera Masumi's Docs. Available at: https://seramasumi.github.io/docs/Tech%20Knowledge/The%20four%20principles%20of%20OOP.html.

Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R. and Yelick, K. (1997). *Processor-Memory Performance Gap.[Hen96]. IEEE Micro*, pp.34–44. doi:https://doi.org/10.1109/40.592312.

PCMag (no date). *Definition of MPSoC*. [online] PCMAG. Available at: https://www.pcmag.com/encyclopedia/term/mpsoc#:~:text=(MultiProcessor%20System%20On%20Chip)%20A.

Prakash, A., Amrouch, H., Shafique, M., Mitra, T. and Henkel, J. (2016). Improving mobile gaming performance through cooperative CPU-GPU thermal management. *Proceedings of the 53rd Annual Design Automation Conference*. [online] doi:https://doi.org/10.1145/2897937.2898031.

Selvakumar, S. (2017). An Insight into Programming Paradigms and Their Programming Languages. *Journal of Applied Technology and Innovation*, 1(1), pp.37–57.

Shopf, J., Barczak, J.D., Oat, C. and Tatarchuk, N. (2008). March of the Froblins. *International Conference on Computer Graphics and Interactive Techniques*. doi:https://doi.org/10.1145/1404435.1404439.

Silva, P.V. and Villela, S.M. (2017). *Applying pathfinding techniques on the development of an Android game*. [online] Available at: https://www.sbgames.org/sbgames2016/downloads/anais/157460.pdf [Accessed 11 Apr. 2024].

Smith, G. (2018). *We Love Performance! How Tic Toc Games Uses ECS in Mobile Puzzle Games - Unite LA*. [online] www.youtube.com. Available at: https://www.youtube.com/watch?v=snU74KXixoY [Accessed 9 Apr. 2024].

Stroustrup, B. (1993). The Early History of Smalltalk. *ACM SIGPLAN Notices*, 28(3), pp.271–297.

Stroustrup, B. (1997). *The C++ Programming Language*. 3rd ed. Reading, Mass.: Addison-Wesley.

Toftedahl, M. (2019). *Which are the most commonly used Game Engines?* [online] Game Developer. Available at: https://www.gamedeveloper.com/production/which-are-the-most-commonly-used-game-engines-#close-modal.

Turpeinen, M. (2020). *A Performance Comparison for 3D Crowd Rendering using an Object-Oriented system and Unity DOTS with GPU Instancing on Mobile Devices.*

United Nations (2021). *Summary of the Asia-Pacific Countries with Special Needs Development Report 2021: Strengthening the Resilience of Least Developed Countries in the Wake of the Coronavirus Disease Pandemic*. [online] Available at: https://www.unescap.org/sites/default/d8files/event-documents/ESCAP_77_4_E.pdf.

Unity (no date). *DOTS - Unity's new multithreaded Data-Oriented Technology Stack*. [online] unity.com. Available at: https://unity.com/dots.

Unity (2020a). *Entities | Entities | 0.17.0-preview.42*. [online] docs.unity3d.com. Available at: https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/ecs_entities.html.

Unity (2020b). *Systems | Entities | 0.17.0-preview.42*. [online] docs.unity3d.com. Available at: https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/ecs_systems.html [Accessed 7 Apr. 2024].

Unity (2020c). *Unity - Manual: What is multithreading?* [online] docs.unity3d.com. Available at: https://docs.unity3d.com/2019.3/Documentation/Manual/JobSystemMultithreading.html [Accessed 9 Apr. 2024].

Unity (2023a). *About Burst | Burst | 1.8.13*. [online] docs.unity3d.com. Available at: https://docs.unity3d.com/Packages/com.unity.burst@1.8/manual/index.html [Accessed 9 Apr. 2024].

Unity (2023b). *Managed components | Entities | 1.0.16*. [online] docs.unity3d.com. Available at: https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/components-managed.html [Accessed 9 Apr. 2024].

Unity (2024). *Unity - Manual: Jobs overview*. [online] docs.unity3d.com. Available at: https://docs.unity3d.com/Manual/job-system-jobs.html.

Vacheresse, K. (2023). *Improving job system performance scaling in 2022.2 – part 2: Overhead*. [online] Unity Blog. Available at: https://blog.unity.com/engine-platform/improving-job-system-performance-2022-2-part-2 [Accessed 9 Apr. 2024].

Van-Roy, P. and Haridi, S. (2004). *Concepts, techniques, and Models of Computer Programming*. Cambridge, Mass.: Mit Press.

Wißner, M., Kistler, F. and André, E. (2010). Level of Detail AI for Virtual Characters in Games and Simulation. *Lecture Notes in Computer Science*, pp.206–217. doi:https://doi.org/10.1007/978-3-642-16958-8_20.

Zulhusni, M. (2023). *mobile gaming 2023: why is mobile dominating all other platforms?* [online] Tech Wire Asia. Available at: https://techwireasia.com/10/2023/what-is-the-state-of-mobile-gaming-2023/.

Ludography

Sunblink (2022) *HEROish* [Video Game]. Sunblink. Available at: https://apps.apple.com/us/app/heroish/id1524305321

University of Applied Sciences Europe
Iserlohn · Berlin · Hamburg

EIGENSTÄNDIGKEITSERKLÄRUNG / STATEMENT OF AUTHORSHIP

Berberov	Alan
Name | Family Name	Vorname | First Name

48831137	Performance Comparison: OOP versus DOTS Paradigms in Mobile Games Made with Unity Engine
Matrikelnummer | Student ID Number	Titel der Examsarbeit | Title of Thesis

Ich versichere durch meine Unterschrift, dass ich die hier vorgelegte Arbeit selbstständig verfasst habe. Ich habe mich dazu keiner anderen als der im Anhang verzeichneten Quellen und Hilfsmittel, insbesondere keiner nicht genannten Onlinequellen, bedient. Alles aus den benutzten Quellen wörtlich oder sinngemäß übernommen Teile (gleich ob Textstellen, bildliche Darstellungen usw.) sind als solche einzeln kenntlich gemacht.

Die vorliegende Arbeit ist bislang keiner anderen Prüfungsbeh.rde vorgelegt worden. Sie war weder in gleicher noch in ähnlicher Weise Bestandteil einer Prüfungsleistung im bisherigen Studienverlauf und ist auch noch nicht publiziert. Die als Druckschrift eingereichte Fassung der Arbeit ist in allen Teilen identisch mit der zeitgleich auf einem elektronischen Speichermedium eingereichten Fassung.

With my signature, I confirm to be the sole author of the thesis presented. Where the work of others has been consulted, this is duly acknowledged in the thesis' bibliography. All verbatim or referential use of the sources named in the bibliography has been specifically indicated in the text.

The thesis at hand has not been presented to another examination board. It has not been part of an assignment over my course of studies and has not been published. The paper version of this thesis is identical to the digital version handed in.

12/04/2024 Berlin	[illegible]
Datum, Ort | Date, Place	Unterschrift | Signature