Master Thesis

Developing Flexible and Scalable Games using Data-Oriented Design in Comparison to Object-Oriented Design

Authors

Frederik Mads Pil (frpi@itu.dk)

Supervisors

Main-Supervisor - Mathias Boldsen Jensen (boje@itu.dk)

Co-Supervisor - Paolo Burelli (pabu@itu.dk)

Stads Code: KISPECI1SE

June 2023

Abstract

Since the 1970s, game development has undergone significant evolution and growth in popularity. This has increased user expectations leading to rapidly increased software complexity and higher demand for the hardware platforms. Balancing appealing games with flexibility and scalability has become a major challenge in the industry. For a long time, Object-Oriented Design (OOD) has been the dominant design paradigm, valued for its flexibility and real-world reflection. However, OOD is now facing scalability issues due to the increasing player demands, requiring more processing of data. In response, Data-Oriented Design (DOD) has emerged as a promising alternative that has gained more attention from developers over the last decade. By focusing on the separation of data and transformations, DOD provides an alternative design approach that leverages modern computer architecture for cache-friendly data layouts, parallelism, and optimized memory access patterns. This maximizes performance and encourages greater flexibility by decoupling the game logic.

This thesis explores the benefits and challenges associated with DOD in game development, specifically in terms of flexibility and scalability. Through interviews with industry experts and a conducted case study, we demonstrate the effectiveness of applying DOD in game development. The results show that both OOD and DOD have challenges in terms of flexibility, specifically both implementations showed a tendency to develop low cohesion and maintainability as more complex features were added. However, DOD promoted greater decoupling between the game features, enabling developers to make modifications more efficiently. Additionally, DOD has proven to provide significant scalability benefits compared to OOD. As the performance gap between CPU and memory is expected to persist, the scalability benefits of DOD are expected to only get more relevant in the field of game development.

Keywords: Data-Oriented Design, Object-Oriented Design, Flexibility, Scalability, Maintainability, Performance, Entity Component System

Contents

1 Introduction

In the continuously evolving field of game development, the demands placed on game mechanics, functionality, and gameplay have experienced exponential growth since the beginning of video games[3]. Creating games that are visually compelling while also being flexible and scalable has become a critical challenge within the industry[27]. Object-Oriented Design (OOD) has long been the dominant software design paradigm, valued for its flexible capabilities and its alignment with real-world reflection. However, it is now facing scalability challenges due to the increased demands from players[27]. In response, Data-Oriented Design (DOD) has emerged as a promising alternative to OOD[27].

DOD shifts the focus from modeling game entities as objects to organizing data processing more efficiently for modern computer architecture. By leveraging cache-friendly data layouts, parallelism, and optimized memory access patterns, DOD aims to maximize performance and scalability. Furthermore, it advocates for the separation of data and transformations, enabling the creation of high-performance gameplay systems while promoting a modular approach that facilitates flexibility in game development[8].

This thesis aims to explore the benefits and challenges associated with adopting a DOD approach in game development, specifically in terms of flexibility and scalability. Through an in-depth analysis of current literature, interviews with industry experts, and the execution of a case study, this research seeks to highlight the advantages that DOD can bring to game development.

1.1 Research Question

The research question this thesis aims to answer is therefore: What are the factors that contribute to the development of flexible and scalable games using data-oriented design in comparison to object-oriented design, and how can these factors be optimized for efficient game development?

1.2 Thesis Objective

This thesis encompasses two primary objectives. Firstly, it seeks to provide a broad historical overview of the evolution of game development methodologies. It focuses on the transition from what some consider a data-oriented approach[15] to the dominant OOD software paradigm we see today in the gaming industry. Additionally, the thesis explores the resurgence of data-oriented ideas, which have gained increasing popularity in modern game development.

The second objective of this thesis is to address the notable gap in the existing literature concerning the impact of DOD on flexibility and scalability in game development. To bridge this gap, this study aims to investigate the use of DOD by conducting a comparative analysis of these two metrics between an OOD and

DOD approach. By examining the effects of these design paradigms on flexibility and scalability, the thesis aims to provide insights into their importance in game development.

1.3 Thesis structure

First the reader will be introduced to the principles of making games in OOD and DOD in Section 2. Next, a historical overview of the events that led to the increased popularity of DOD is provided in Section 3. Next, the related work already performed on the topic of flexibility and scalability will be discussed in Section 4. In Section 5 a series of conducted interviews with industry experts on the current use of DOD will provide real-world insight into the use of DOD. Section 6 introduces the conducted case study while Section 7 explains the evaluation of the results from the study. The results will be presented in Section 8 and discussed in Section 9. Finally, future work is proposed in Section 10 while Section 11 concludes the findings.

2 Preliminaries

In this section, the fundamentals of OOD and DOD are explained. The difference in flexibility and scalability in the two design paradigms are discussed by examining common design strategies, memory layout and memory-access patterns.

2.1 Flexibility and Scalability

The thesis centres on the concepts of flexibility and scalability in game programming. This section aims to provide precise definitions for these terms.

Flexibility, refers to the ability to promptly and effortlessly modify the design or architecture of a game without necessitating extensive rework of existing code. A flexible game is characterized by its ability to accommodate changing requirements without affecting other components within the game[31].

Scalability, denotes the game's capability to manage augmented volumes of data and computational processing as the game expands. A scalable game possesses the capacity to meet the demands of growth without compromising its performance[31].

2.2 Object-Oriented Design

Object-oriented programming (OOP) presently stands as the dominant programming paradigm within the domain of game development. It is a challenging endeavour to locate a game that does not regard its content as a collection of interacting objects, constructed through classes[27]. Similar to other facets of

software development, game development is an iterative process, containing numerous iterations that often span several years. The initial game design undergoes substantial transformations throughout its development, making the game barely recognizable in its final implementation caused by numerous requirement changes during the development stages. Therefore, game development necessitates a flexible design approach to cope with these changes, while minimizing the impact on development time.

OOD promotes modularity and encapsulation. This enables game developers to organize the codebase into reusable and self-contained objects, encouraging better maintainability and extensibility[31]. This modular design approach eases the management and evolution of game projects over time. Concepts such as abstraction, polymorphism and inheritance are commonly used strategies within software development to create flexible systems[5]. In OOD, code reusability is achieved through these concepts. Inheritance enables developers to derive new classes from existing base or parent classes. Through this, derived classes inherit the data and behavior encapsulated in the base class. It is a concept used for code reuse by eliminating the need for duplicating common functionalities. This is highly relevant in game development since games often feature objects with common logic such as variations of enemies that all need the same movement logic but have slightly different attack variations[31]. Developers can extend the base classes to add or override specific features, hence modifying the inherited classes to the specific needs.

The abstraction and reusability aspects of OOD provide a number of benefits for game development. They enable designers to create modular and interchangeable components, reducing code duplication while improving maintainability throughout the evolution of the project. When abstracting common data and behavior developers can create flexible games that can accommodate requirement changes[31].

In OOD, cohesion and coupling are important concepts. Cohesion refers to the relationship and interdependency between functionalities within a class[14]. It is used to evaluate how well the functionalities and responsibilities of a class align with each other. Maintaining high cohesion is important to make the codebase manageable and organized. Developers can achieve greater code flexibility by making sure the components of a class are closely related and have a well-defined responsibility. In software development this is often referred to as the *Single Responsibility Principle* (SRP)[23]. By adhering to the SRP, developers strive to implement code that is more flexible.

On the other hand, coupling is a measure of dependencies between classes. It is used to determine how closely two components rely on each other[14]. With high coupling, the code is more difficult to modify, as changes in one part might affect other parts, forcing developers to spend more time on changes. The concept is important in game development to achieve a flexible design. Using abstraction to define common behavior decouples the specific implementation,

which allows for easier code changes and the extensibility of classes that share common behavior.

Implementing flexible code is an important aspect of game development[7]. However, flexibility often implies trade-offs on scalability in OOD. While OOD promotes encapsulation, modularity and code reusability, these benefits can potentially impact performance. The use of objects and classes introduces additional memory and processing overhead[14]. An object comes with its own set of data and functionalities that requires memory allocation. It is common to experience memory fragmentation and scattered allocation when dealing with OOD[32]. Let's consider a simple example in which three enemy types are spawned and allocated in memory.

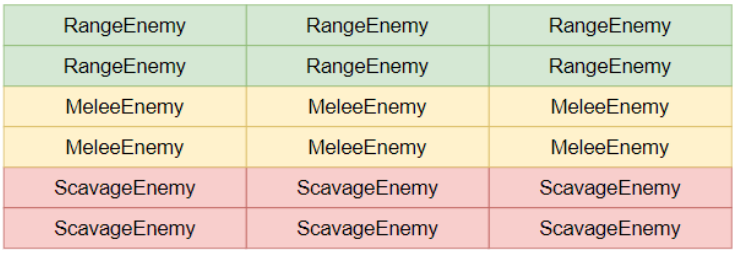

Figure 1: Simple illustration of memory layout after spawning 18 enemies of three different types.

Figure 1 depicts a simple memory layout after spawning 18 enemies of type RangeEnemy, MeleeEnemy, and ScavageEnemy. While the memory is organized in this situation it will quickly become fragmented and scattered when more enemies are created and destroyed.

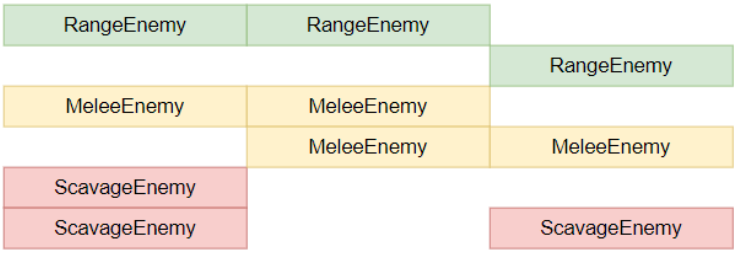

Figure 2: Simple illustration of memory layout after creating and destroying enemies during play. The memory layout is now scattered and fragmented.

Figure 2 shows the memory layout after creating and destroying enemies randomly during gameplay. When memory fragmentation becomes significant, it leads to inefficiencies in memory allocation and memory access that can impact the game's overall performance. Developers must be aware of these limitations.

Additionally, the use of inheritance can introduce further performance costs. When a derived class calls a method that is defined in its base class the system must first go through the inheritance hierarchy to find the correct method to call[2]. When a class is derived from a base class that has virtual functions, the derived class inherits the *v-table* of the base class as well. This is a data structure that contains pointers to the virtual functions of each class. When the derived class overrides a virtual function, it replaces the function pointers in the v-table with pointers to its own implementation. The use of v-tables in class inheritance introduces performance overhead due to indirection. The implementation needs to perform an additional level of indirection to access the correct function in the v-table which introduces a small computational cost compared to direct function calls[2]. This becomes problematic in game development since a game often features many of these small costs which potentially will result in a larger overall performance decrease.

2.3 Data-Oriented Design

DOD is an alternative software engineering approach that can aid some of the challenges of OOD. The primary objective is to clearly separate data and transformations[19]. Unlike OOD, which encapsulates data and logic within objects, DOD employs memory-efficient layout patterns to facilitate contiguous data storage. This optimizes memory access time during data transformation[19].

Instead of structuring data as components of objects, DOD arranges data based on how and when it is accessed. Storing data that is frequently accessed together in memory improves memory access efficiency, leading to improved performance, especially in the context of game development[27].

In DOD, the tearing apart of encapsulation and inheritance promotes a more direct approach to data transformation. Developers are forced to shift the focus from objects to data optimizations. Efficient use of data structures such as arrays aims to maximize data locality and cache utilization to minimize the impact of cache misses[19].

2.3.1 Entity Component System

ECS has become the main design pattern in DOD, some even confuse the two terms thinking that ECS is a synonym for DOD[35]. This section describes the fundamentals of ECS which is the chosen framework used for the case study in this thesis.

The objective of ECS is to decouple objects into separated data and logic [21]. Through this separation, ECS achieves a modular design, which allows for memory-friendly data storage[27]. This is done by organizing data and logic into three different units: *Entities*, *Components*, and *Systems*.

Entities: are the fundamental building blocks of the objects. An entity represents a concept or an object on an abstract level since they do not hold any data or logic of an object itself. Entities are simply just unique identifiers which distinguish entities from each other[38]. Its only function is to group together related components that form it into a representation of some object. This is done by marking the components with the entity identifier to indicate what entity they belong to[21].

Components: are small, self-contained pieces of data that represent a specific property of an entity. They contain all the data belonging to their respective entities. Components are often designed to be reusable in order to be attached to multiple entities[21].

Systems: are responsible for performing transformations on the data stored in the components of specific entities. A system defines the logic that acts upon entities with a certain combination of components. Typically, a system iterates through all entities with the specific set of components and performs the same operations on the data[38].

Using an ECS implementation can improve the flexibility of the game design. While OOD typically revolves around classes and their inheritance hierarchies, ECS focuses on composing entities from individual components and systems[27]. OOD can lead to complex inheritance, low cohesion and high coupling if not considered carefully, especially for bigger complex games. The component-based approach in ECS allows for greater flexibility since data and operations are decoupled without the need for complex class hierarchies[3].

2.4 CPU and Memory

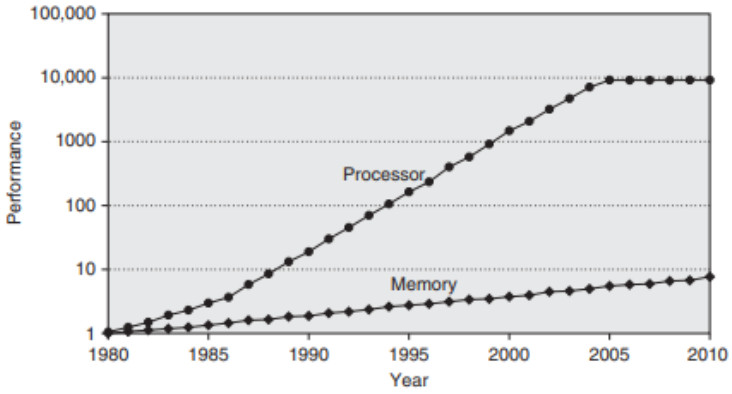

Figure 3: Development of processor and memory performance throughout three decades since 1980. The performance is measured as the difference in the time between processor memory requests and the latency of a DRAM access [22].

The main challenge of achieving high performance in modern computer architecture is the performance gap between microprocessors (CPU) and memory access speed. Since the early days of personal computers, this performance gap has been increasing exponentially leaving memory access speed far behind modern multi-core CPUs. This increasing processor-memory performance gap has now become one of the primary challenges of increasing computer performance because the CPU is able to process instructions significantly faster than data can be accessed from memory, leading to the so called *Von Neumann bottleneck* [10]. In the Von Neumann CPU architecture, instructions and data are stored in the same memory space, and the CPU needs to access this memory sequentially. This means that the CPU cannot simultaneously access memory or perform computations while the CPU is fetching an instruction from memory[10]. This can significantly impact the overall performance of an application, especially for applications with large amounts of data.

To accommodate this gap, modern computer architecture makes use of *caches*. The cache is a smaller memory located closer to the CPU, which stores frequently accessed data in contiguous blocks to improve accessibility time. When a processor accesses data in memory, it reads the data from memory into a cache before doing transformations on the data, this is called a *cache line*. The cache line is a block of memory which is loaded into the cache at the moment when data is requested from memory. Typically, a cache line contains 64 bytes, which means that 64 bytes of memory are always loaded into the cache when some data is requested, even if only a single byte is requested[19].

The cache line is important when considering the performance of CPUs and

is often mentioned in literature about DOD[27]. This is because data access directly from memory is much slower than reading data from the cache.
A common way to lay out data in DOD is to store the data in arrays to improve the processing of larger amounts of data. This is performed to store the data in contiguous memory locations in order to utilize the cache as much as possible. Storing data this way makes it more likely to be loaded into the cache in fewer cache lines, minimizing the number of cache misses, and hence improving the performance[8].

2.4.1 Memory allocation

In OOD, data is grouped by objects in memory using an approach called *Array Of Structures* (AOS)[43]. AOS represents objects as contiguous blocks of memory, where each block contains the data of an object. This results in all data instances of a class being stored together one after another in memory.

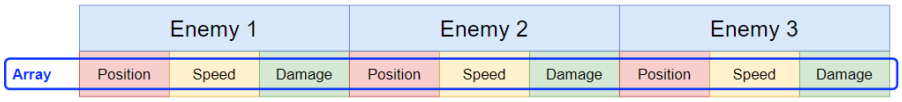

Figure 4: Three enemy objects stored in memory as one array.

Figure 4 shows an example of three enemy objects stored in memory. Each enemy contains data for position, speed, and damage. In an AOS style, all of these enemy properties are stored in one contiguous array. Using this memory layout simplifies memory management. Objects of the same class are stored contiguously, hence making it easier to allocate and deallocate memory. In OOD, memory allocation and deallocation are often handled by mechanisms built into the language using keywords like *new* and *delete* with constructors and deconstructors. Objects encapsulate both the data and transformations, allowing for simple memory management since memory is tied to the lifetime of the objects[20]. Furthermore, deallocation is often handled through automatic *garbage collection* which is performed once the deconstruction of an object has been performed. While this memory management is convenient to the developer it comes with a cost. AOS can lead to poor memory access patterns when accessing the data. Due to the data layout, accessing different properties of the same object requires jumping around in memory which can lead to an increased number of cache misses[27]. For example, if we want to update the position of all enemies, we have to jump around memory when accessing the data because the speed and damage data are not used. The basic constraint with AOS and memory access is the *cache line*. When the position needs to be updated for the enemies, the cache line will be loaded with 12 bytes for the position data (x, y, and z data), 4 bytes for the speed, and 4 bytes for the damage, in total 20 bytes for each single enemy object until the cache line is full. In this case, the cache line would contain 8 bytes of data for each enemy, that are not used

for any calculations. This results in a number of cache misses when iterating through hundreds of enemies.

In contrast, DOD typically involves an increased manual memory management approach. DOD focus on optimizing data layout and memory access patterns to achieve better performance. The goal is to maximize *data locality* and minimize cache misses[19]. Data locality refers to the concept of organizing data in a way that maximizes the efficiency of memory access. This involves keeping related data sequentially in memory to minimize the number of jumps needed to access the data[25]. Data locality includes two main types: *temporal* and *spatial* locality.

Temporal locality is the concept of keeping data in memory that is frequently accessed. By keeping frequently accessed data in memory, the data access time is decreased once accessed again.

Spatial locality refers to the concept of organizing related data contiguously to allow for efficient retrieval of multiple data elements.

DOD utilizes *Structures Of Arrays* (SOA) instead of AOS to improve data locality, spatial locality, and temporal locality[8]. In SOA, instead of storing the parameters of an object in a single array, the data is stored in separate arrays, each containing a single property.

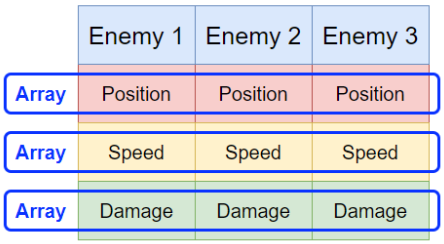

Figure 5: Three enemy objects stored in memory as three separate arrays.

Figure 5 shows the same example with enemies but instead uses SOA for the data layout. We now have three separate arrays for storing position, speed, and damage data resulting in contiguous storage of the same data type[25]. Now if we want to update the position of all enemies, a cacheline containing only the position data is pulled to the cache. As a result, we greatly minimize the number of cache misses because only relevant data is stored in the cacheline. From the memory point of view, there is no such thing as an enemy object anymore instead separate collections of data representing enemies are present. This complicates memory management for the developer since all separate arrays have to be modified when creating or destroying enemies. Memory allocation and deallocation have to be done manually by the developer which requires the developer to keep a reference to all data entries in the arrays that hold data for an enemy[19, 27].

In game development, we still want to keep the idea of objects containing different data and behaviors, but at the same time optimize the memory layout to gain performance benefits. The most common way this is solved in DOD is through the ECS design approach. Entities can be viewed as an object that encapsulates components together, to form an abstract representation of an object. While systems perform logic operations on the data stored in the specific sets of components.

3 Historical overview

The term, DOD, has been present for a couple of years first mentioned by Noel Llopis [27] in 2009, defining DOD as a software paradigm that shifts the focus perspective of programming practices from objects to the data. "*Programming, by definition, is about transforming data: It's the act of creating a sequence of machine instructions describing how to process the input data and create some specific output data.*"[27].

Ever since, this definition has been questioned and challenged especially within the games industry[15]. Many would argue that DOD has been around since the birth of video games and home consoles, where some of the same practices were used in order to create executable games targeted at the limited hardware available at the time.

Taking a look at the very beginning, many people would consider "Pong"[6] to be the first real electronic video game, released in 1972 by Atari which became extremely popular amongst people and aided the launch of the video game industry[40, 39]. The original Pong game had a very simple design compared to modern video games. It consisted of two paddles that players controlled, in order to bounce a ball back and forth between each other.

Games like Pong, and others, made before personal computers and game consoles were made with specific hardware for the specific games. This meant that players would have to buy a new piece of a machine containing the specific game they wanted to play[39]. In the 1970s the relationship between console manufacturers and game developers where established in order to decrease the amount of work required by manufacturers and increase the number of available games to the public[33].

During the 1970s and 1980s home consoles came to life with the most popular one being Atari 2600 released in 1972. At that time, the hardware components were nowhere near what we know today. Atari 2600 only featured an 8-bit processor and was equipped with only 128 bytes of memory that were used to store all state variables in the games[33]. Games had to be stored externally on storage devices called cartridges which would be inserted into the console. The cartridges had a built-in 6-kilobyte ROM which contained the entire game code[39].

Due to these hardware limitations of the Atari 2600, the game design was

not very flexible and hardly possible to maintain. Game programmers had to customize the game code to be executable on the constrained hardware. Any little change to the codebase could have a significant impact on both the performance and functionality of the game, which made it very time-consuming to maintain the code. In most cases, the games were programmed with machine-specific programming languages like assembly and targeted for the hardware, resulting in poor flexibility. Code reuse was almost nonexisting because each game-specific feature had to be coded onto the memory layout[33].

One could possibly argue that Atari developers at the time created games in a data-oriented way without being aware of the term. Dino Dini argues in his blog post "Beam me up, Scotty!"[15], that DOD is nothing new but simply the act of building programs where data organisation is fundamental to efficiency for both memory capacities and speed. He argues that this is what Atari developers had to deal with at the time. They would come up with all sorts of hacks and tricks in order to make the games run on limited hardware.

A commonly used trick to minimize the workload for the CPU was frame skipping. Instead of rendering the game every frame, developers would program the logic to render every second. This was possible because the graphics renderer and game logic were separated. The game logic would still be calculated correctly, completely independent of the graphics. The downside to this approach was of course the visual representation of the game. Players could experience flickering or stuttering which can be annoying to the players[39].

In 1984, Shigeru Miyamoto and his team published the game "Excitebike"[29], a side-scrolling 2D racing game. The game was programmed using the assembly language due to hardware constraints. The interesting part is that they managed to create a smooth side-scrolling mechanic which could be reused for their later published title "Super Mario Bros"[30], which featured the same side-scrolling code implemented in Excitebike[44].

The time of assembly programmed games continued for some time but eventually, a gradual shift to high-level programming languages such as C and C++ happened. To minimize development time, developers sought more flexible solutions. Assembly is a low-level programming language that directly manipulates machine code. It benefits from being highly efficient since developers can directly manipulate the memory stack, but at the same time difficult to maintain, especially for complex games[33].

Alongside the evolution of more complex games, the need for flexibility became noticeable. High-level programming languages provided an interface to the hardware, making it easier for developers to write and maintain the games. Object-oriented programming would prove to provide the ability to define custom data types and methods that could be reused throughout the codebase, greatly reducing code duplication[39].

In the mid-1990s, OOD became a significant part of game development. As

games became more complex, developers started searching for high-level programming languages that could ease the development process. C++ became one of the first languages used by game developers due to its object-oriented features which made it easier to create complex game systems[44]. C++ was used to create classes that represented game objects such as characters, enemies, and items. The use of an object-oriented language like C++ provided a higher level of abstraction, allowing developers to write code that was more maintainable and readable. The use of classes, inheritance and polymorphism enabled developers to organize code into reusable components in order to create complex game systems more efficiently. At the time, the productivity gains from using OOD outweigh the performance gains from coding assembly games[39].

In the 2000s, C++ gained more popularity due to the increased popularity of modern console gaming like PlayStation and Xbox. This required developers to make games that could be played on a variety of different consoles. C++ provided a high degree of platform independency, making it possible to create games for multiple consoles at once[40].

In 2006, PlayStation released their third edition of the popular gaming console. The architecture of this console was significantly different from previous gaming consoles[12]. While the PlayStation 2 featured single-core processing, PlayStation 3 featured multi-core processing which allowed for parallel processing and task distribution[1]. Game developers had to adapt their programming practices and optimize their code to take full advantage of the new hardware capabilities. OOD often suffered performance issues on this architecture due to the inefficient memory access patterns and the overhead of managing object-oriented structures[12, 27].

DOD, on the other hand, focuses on data organization which maximizes memory locality and facilitated better parallel processing[27]. While OOD relies on class hierarchies and virtual function calls, DOD emphasizes SOA data layouts to allow for better cache utilization and improved memory access time[1]. By programming data-oriented, developers could leverage the parallel processing capabilities of the new hardware more efficiently.

Until most recent years, rapid growth has happened in the video game industry, starting from those simple assembly-programmed games like Pong. The demand of requirements in video games has seen a drastic change from those simple 2D games with few interacting objects and hardly any complex graphics, to what we see player demanding today. Massive-scale multiplayer, open-world, hyper-realistic, and real-time physics-based games are common requirements nowadays. So far, the video game industry has been able to catch up with these requirements due to the evolution of hardware. Unfortunately, that evolution has only slowed down with memory access speed not keeping up with processor performance while player demands only seem to grow ever higher[11]. OOD is limited by the performance because we ignore cache lines and how modern CPU architecture works[8].

With the increased demand for games requiring the processing of ever more data and the limitation of OOD, developers started searching for more cache-optimized approaches to create games. While some games utilized data-oriented principles to optimize performance, DOD first gained momentum with the publications of the article "*Data-oriented design (or why you might be shooting yourself in the foot with OOP)*"[27] by Noel Llopis in which the author points out the scalability and flexibility benefits of programming games in a data-oriented manner[27].

4 Related Work

The field of game development has witnessed continuous evolution throughout the years to address the challenges of creating flexible and scalable games. In recent years, a boom in the data-oriented design paradigm has happened. More developers have opted for data-oriented techniques and design patterns to create flexible and scalable games that can keep up with today's user requirements. This section provides an overview of existing litterateur related to the case study performed in this thesis.

4.1 ENCODE

While numerous studies have explored scalability in DOD game development, few have explored its capability of flexibility. Anne van Ede[43] investigated conversion methods to convert OOD games to DOD. This study implemented an automatic conversion tool to ease the work of developers while acting as a learning tool for new developers getting into DOD. The study conducted in-depth interviews with multiple industry experts on DOD. The results showed that the DOD approach, ECS, was often chosen for its flexibility and maintainability benefits rather than its performance possibilities. The experts argued that having larger components, hence resulting in fewer components, decreased the complexity of the code, making it easier for collaboration amongst team members because the code was more readable.

4.2 Application of DOD

K Fedoseev et al.[3] investigated the impact of OOD and DOD on performance in game development. This study utilized a comparative approach to evaluate the performance and maintainability of two games developed with OOD and DOD principles. The results revealed that the data-oriented approach consistently outperformed its object-oriented counterpart in terms of frame rate, memory usage, and CPU utilization. They attributed the performance difference to the data access patterns and cache locality achieved when applying data-oriented principles.

Furthermore, the study also explored the maintainability aspects of DOD in comparison to OOD. They concluded that both approaches had a similar level of maintainability but that it required more time for developers to modify the DOD codebase. This was addressed to the two development teams which both had prior experience with OOD but not with DOD. Therefore, the complexity of DOD required more knowledge gathering by the team implementing the DOD version, before the features could be implemented. However, the results on maintainability were mainly concluded based on static metrics calculations while no analysis on the structure of the codebase was performed. This do not provide a complete picture of the maintainability level of the game.

4.3 CPU and Memory Performance in Mobile Games

Björn Eriksson and Maria Tatarian conducted a study to investigate the CPU and memory performance differences between OOD and DOD in mobile games[17]. They implemented two identical mobile games, one with OOD and the other with DOD to collect their empirical data. The results showed that the CPU usage was significantly different for games with large amounts of data. The DOD version clearly outperformed the OOD version where the CPU spends 20.9% of the time on updating data for the DOD version while the similar setup for the OOD version spends 69,2% of the time on the same action.

However, the game design implemented was very simple and do not imply the exploration of interacting objects in a real-world game scenario. They created a simple game consisting of spawning cubes that simply rotated and elevated. While this setting can be used to monitor the workload of the CPU, it does not incorporate features in which game objects interact or receives data from others which also have a significant impact on CPU performance.

5 Flexibility and Scalability in Games

The field of game development is in constant evolution requiring ever more features from craving users. DOD has emerged in the past couple of years as a promising approach to building efficient and scalable games. However, current literature is very limited on the topics of flexibility and scalability in DOD for games and there is a lack of consensus on the best practices for designing data-oriented games.

To address the gap in the literature, a series of interviews has been conducted to gather insight from experts in the field of DOD. By conducting these interviews, we hope to gain a better understanding of the limitations and benefits of designing data-oriented games in terms of flexibility and scalability.

5.0.1 Target Group

The interviewees were selected based on their knowledge of game development in a data-oriented manner. Most of the participants either worked or had previously worked within a game development company utilizing some data-oriented framework. Some of the participants even contributed to the development of the data-oriented framework developed by Unity. Additionally, all participants had more than 10 years of experience within the field of software and game development.

5.1 Interview Setup

Five experts with knowledge of DOD in game development were interviewed. All participants were interviewed in person. After a short introduction to the project, they were asked 16 open questions. The following section describes how the interviews were conducted.

5.1.1 Question Topics

The 16 open questions were divided into four categories:

Personal Experience: These questions aimed to gather information regarding the interviewees' personal experiences with DOD, in order to validate their authority and contextualize the topic. The collected data will be used to shed light on the current grey area surrounding the definition of DOD in the literature. Understanding how the interviewees define DOD can aid in categorizing their responses and further contribute to the knowledge in this field:

1. How would you define data-oriented design?
2. Can you tell me shortly about what initially got you started with data-oriented design?
3. Can you tell me how you are currently working with data-oriented design?
4. Can you tell me what data-oriented design pattern and/or programming language you are currently working with, and why that specific one?
5. In your view, what is the most significant advantage of data-oriented design, summarized in a sentence or two?
6. In your view, what is the most significant disadvantage of data-oriented design, summarized in a sentence or two?
7. In your opinion, what types of games do you think benefit from a data-oriented design approach?

Scalability: The interviewees were questioned on the game development process, with an emphasis on performance optimization. These questions aimed to

gather information about their process and way of thinking when creating new systems, as well as their approach to structuring data in an efficient manner in order to achieve optimal performance:

8. What do you focus on when designing a scalable feature or system?
9. How do you ensure the scalability of your data-oriented designs, and how do you measure their effectiveness?

Flexibility: Questions in this category focus on the flexibility of the codebase when developing games in DOD. Specifically, the focus lies on gathering information regarding the interviewees' strategies for developing and modifying existing code to accommodate requirement changes:

10. How does using data-oriented design impact your ability to handle changes in game requirements and design?
11. To what extent does data-oriented design enable you to experiment and iterate quickly?
12. Follow-up: How does that compare to other ways of programming, like object-oriented programming?
13. Could you describe the impact that the use of data-oriented design has had on collaborative work?

Closing: Lastly, the interviewees were asked some closing questions, to gather any important information which they felt was not covered in the previous questions:

14. Could you describe any cases where DOD would impede the flexibility and scalability of a project?
15. What do you think the future of DOD in game programming is? Is it here to stay?
16. Is there anything you consider important about the use of data-oriented design that I have not asked you about?

5.2 Results

This section will summarize the results gained from the conducted interviews. The results have been grouped based on the four categories. The complete transcripts can be seen in Appendix **??**.

5.2.1 Personal experience

When questioned to define DOD, most of the participants mentioned the importance of understanding the relationships between data and hardware in order to optimize cache utilization. The interviewees argued that one must understand the workings of computers in order to create high-performance game code.

Interviewee 1: *I think of DOD as putting the data first. Not thinking in terms of behaviour, objects, and messaging. Making data the focus of your design work.* (See Appendix **??**)

Interviewee 2: *I think it is something knowing how memory and cache work. That is basically it. You need to know how computers work and the hardware you are working with.* (See Appendix **??**)

Interviewee 3: *... First thing is related to performance, the idea of data structures of algorithms are being selected or designed from a desire of having a better performance in particularly with a focus on data-access patterns.* (See Appendix **??**)

Interviewee 4: *... the fact that being aware of performance and what good performance requires which is knowledge about your hardware, data, and code.* (See Appendix **??**)

Interviewee 5: *I would say that DOD is a programming practice mainly used for performance optimization by focusing on the hardware in which we are processing the data of our programs.* (See Appendix **??**)

Another key similarity between the participants was the argument of when to use DOD. Most of the interviewees mentioned that the advantage of DOD was the performance gain, but the disadvantage was the urge of using DOD to solve everything. They argued that DOD should be used depending on the type of problem it tries to solve, and not for the sake of making a DOD game.

Interviewee 1: *I think it is important to look at the problem that you have and figure out what is the best way to solve that problem rather than just say, this is an entity game so it has to be solved with this technology.* (See Appendix **??**)

Interviewee 3: *The biggest disadvantage is that people become in love with the idea and try to use it for everything. I think that people tend to use it for everything just for the sake of doing it.* (See Appendix **??**)

Interviewee 4: *Also the fact that DOD can almost become a religion. People can fall so much in love with it that they use it for everything also when it is not the best suit for the task.* (See Appendix **??**)

DOD requires more knowledge when developing games, one must know the data and operations performed on that data before deciding on the programming approach. One of the participants mentioned DOD as a dangerous programming approach, it is easy to fall in "love" with DOD and wanting to use it to solve all your problems, but this might make your code more complex than necessary.

Participants were also questioned about how they got into DOD, how they are currently working with DOD, and what they think the future is. An interesting point-of-view which was mentioned during the interview was that most of them got started with DOD because of increasing requirements from players in terms of what games should be, do, and evolve. One of them mentioned how he observed video games becoming ever more complex requiring significantly more processing power from the hardware used to play these games. Due to the limited hardware available to perform all of these tasks, they had to opt for DOD in order to keep up with the user requirements.

Interviewee 2: *I have been doing games for 30 years, and I have been through the period of object-oriented programming came as the big saviour, but after 10 years, suddenly you figure out that OOD is not so good anyways.* (See Appendix **??**)

Interviewee 3: *I would say the thing that got me started was hardware constraints in which it was necessary to optimize everything in order to get what you wanted to run* (See Appendix **??**)

5.2.2 Scalability considerations

To gather information about developing scalable features in games, all interviewees mentioned profiling and testing as the main tools to identify performance-critical implementations.

Interviewee 1: *Performance testing is key. This is a common problem, if all you are doing is testing spinning cubes you are not really testing.* (See Appendix **??**)

Interviewee 2: *I use the profiler all the time and I think that you always have to have a feeling of what you expect from different things in your game, performance-wise.* (See Appendix **??**)

Interviewee 3: *... So I try to pinpoint where I should focus my attention by profiling a lot.* (See Appendix **??**)

They argued that developers must have a good idea of the scale of the features, the data needed, and the transformations needed to be performed, as well as how often and when these transformations are to be executed. Therefore, to measure the effectiveness of the features, performance testing early and often is key to creating a scalable game.

Being aware of the data needed to solve a specific problem was also one of the main topics mentioned when asked about scalability.

Interviewee 4: *I think the most important thing to focus on is the data structure so that it is optimized for performance to minimize memory usage. So for me that would include using the right kind of data.* (See Appendix **??**)

Interviewee 5: *I think the first thing that would be important to focus on is to figure out your data. If you don't know your data model, it is very hard to be able to optimize accordingly to the data.* (See Appendix **??**)

The interviewees argued that developers using DOD must have a deeper understanding of the data that will be utilized in the game. Profiling the solutions can help the developers to make informed decisions regarding how to structure the data to achieve optimal performance. Developers must consider the data structures and algorithms used to transform the data.

5.2.3 Flexibility considerations

The interviews showed that flexibility is often among the reasons for choosing to make a game with DOD. When asked about flexibility most of the participants mentioned that DOD required more development time and felt like a stumbling block when trying to implement something quickly. This made it more difficult and time-consuming in the early phases of development which often takes many iterations in order to settle on some specific feature or game design.

Interviewee 1: *Gameobjects are just way faster to prototype compared to entities.* (See Appendix **??**)

Interviewee 2: *I don't think it does. I think it is worse and I feel that it is the first stumble block when trying to make something quickly.* (See Appendix **??**)

Interviewee 4: *It does require a bit more time in terms of setup and generally you have to write a bit more code in order to get something up and running.* (See Appendix **??**)

It was argued in one of the interviews that developing a game with DOD requires knowing the data beforehand, but that is often difficult, since requirements, and hence data, change during development.

Interviewee 2: *It is knowing your data, and the point is, you don't know your data from the beginning. And making a decision on your data somehow locks you in.* (See Appendix **??**)

The interviews also highlighted a split in the opinions regarding flexibility between the participants. Some of the participants found it difficult to adapt to requirement changes and refactor code in DOD because the structure is often very flat and many components are reused throughout the project, while others viewed this as one of the benefits of DOD.

Interviewee 1: *Comparing DOD and OOD in terms of flexibility they both have their own difficulties with refactoring. I found it difficult to refactor in DOD because the code structure is so flat that you had to go to 20 different files to figure out how to change something.* (See Appendix **??**)

While others see benefits in terms of flexibility when using DOD. To handle changes in game requirements and design, the separation of data and systems was mentioned as an advantage. This decreases the coupling between data and systems. Therefore, changes to one system can be made easier without affecting other systems.

Interviewee 4: *I think the biggest force of applying DOD is that it forces programmers to keep things flat and simple.* (See Appendix **??**)

Interviewee 5: *... But once your implementation starts to increase in complexity I think that DOD has the potential to make it easier to adapt to changes.* (See Appendix **??**)

However, participants with a positive view of DOD in terms of flexibility also mentioned that it takes significantly more development time to get something up and running. But once the initial setup has been done and the implementation starts to get more complex, they thought that adapting to changes in DOD required less work.

When asked about change requirements the interviewees mentioned that the use of DOD highly depends on the use case.

Interviewee 3: *I think this depends on the use case of DOD. It is different if I use DOD to achieve flexibility or performance. Mostly, I have used DOD to achieve performance and from my experience, optimizing things makes it less flexible.* (See Appendix **??**)

The way of programming and structuring the codebase was different whether you programmed for flexibility or scalability. In the interviewees' experience, DOD was mainly utilized for performance gains and was often applied late in the development process because optimization often added inflexibility to the codebase.

5.2.4 Closing

To gather knowledge about the current use of DOD and in the future, the participants were questioned about what type of games they see fit for DOD and what they think the future of DOD in game development was.

Interviewee 1: *... my favourite game is a large-scale simulation game with a lot of stuff happening. In this game, they use gameobjects where that is suited and made their own entity system where that is needed for the high-performance stuff.* (See Appendix **??**)

Interviewee 4: *That would be games that require the processing of very large amounts of data. That could be games like large-scale simulation games.* (See Appendix **??**)

The most important consideration regarding games was that the data transformations should fit a linear data layout meaning games that are able to perform calculations linearly on the data. A specific type of game was not mentioned, but games that require the processing of large amounts of data may benefit from DOD.

When asked about the future of DOD, most participants mentioned that the concept of data-oriented programming has been present for a long time. The problem was that DOD did not have a name at the time. Instead, developers were referencing the likewise concepts as the "cache talk" or "optimization phase".

Interviewee 1: *It has been here forever. What we are thinking of as new now the PlayStation developers already did, but it just did not have a name.* (See Appendix **??**)

Interviewee 2: *I think the problem with the word DOD is that it has become too many things and we should go back to the principle of thinking about games again. The problem is, that OOD came and said, now we don't need to think about data, cache, and memory anymore. But that is not the case, we do need to think about this still.* (See Appendix **??**)

Interviewee 4: *I think because we see this increased new popularity and the fact that hardware is not getting any better there will be much more focus on hardware, performance and data. We still see high and new depends in game development, but we need to optimize all that we can to accommodate these requirements.* (See Appendix **??**)

Some participants argued that this new interest and naming of data-oriented programming was due to the increasing demand of players, that did not follow the current hardware aspects of computers. Optimizing game code has just be-

come more mainstream because it is a necessary aspect of creating games if you want to compete with other game makers.

5.3 Summary

The results from the interviews provided valuable insight into important aspects when developing games using DOD. From this, best practices and guidelines when designing flexible and scalable games can be provided.

The results showed that hardware knowledge is crucial to the performance of games. Developers must have a greater knowledge of the data layout and access patterns when working with DOD. Especially, the separation of data and transformations is essential in DOD when focusing on flexibility and scalability. The results showed that DOD might not always be the perfect fit. When designing a game, the specific task must be considered in terms of data and transformation before choosing to apply DOD.

1. Great understanding of the data and hardware is necessary.
2. If the data is not transparent, DOD might not be the best fit.

When creating scalable games, profiling and testing are important tools to locate bottlenecks in the game design. This can help developers identify issues with the data structure and the transformations being executed.

3. Use profiling tools early and often to locate issues within the data structure.
4. Stress testing should be performed to know the scalability level of the design.

In terms of flexibility, the results showed opposite opinions concerning the potential benefits. Using a component-based approach makes the code more flexible since the behavior of entities can easily be changed by switching out components, but developers must be careful when structuring the project in order not to get lost in the fairly flat structure that tends to appear in DOD.

5. Component-based approaches can improve flexibility and make the code more maintainable.
6. Structuring the codebase is important in order to stay organized.

6 Implementation details

This section introduces the experimental implementation of the conducted case study. In this project, two separate implementations of the same game have been developed. One using the Unity engine[1], and one using the data-oriented

[1] Unity https://unity.com/

technology stack (DOTS)[2] provided by Unity[42]. The aim of this case study is to compare and analyze the flexibility and scalability of the two implementations. By doing so, the thesis aims to evaluate the potential benefits of using DOD in game development and understand its impact on code flexibility and scalability.

The functionality of the two implementations is identical, but one is developed using the traditional object-oriented approach from the regular Unity engine, while the other utilizes Unity DOTS. It is worth mentioning that, at the time of implementing the DOTS version, the BETA version (1.0.0-pre.15) was used.

The source code for all iterations of the two game versions can be found in Appendix **??**.

6.1 Unity DOTS

For the DOD implementation, the data-oriented framework provided by Unity was used. It utilizes an ECS design pattern with *archetypes*. Archetypes in this context refers to a specific combination of component types that entities possess. It works by grouping together similar entities in order to optimize data access and processing for the CPU. When executing different systems we can simply make a query for a specific archetype in which the system will process all entities with that specific set of components[42].

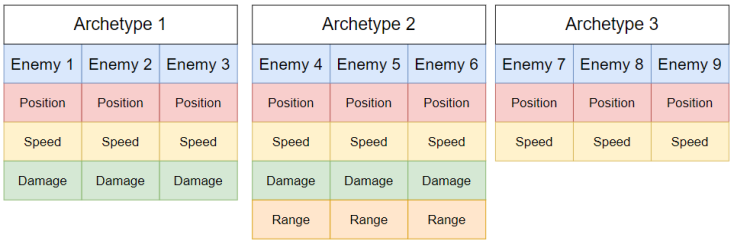

Figure 6: Three different archetypes representing three different enemies with each own set of unique components.

Figure 6 show an example of three different enemies each containing its own set of unique components being grouped into three different archetypes.

Furthermore, Unity DOTS use a special compiler called *Burst*. It is designed to work in conjunction with the ECS framework and job system, all incorporated in Unity DOTS. The Burst compiler takes C# code and compiles it into optimized machine code in order to use *Single Instruction, Multiple Data* (SIMD)[45] when possible. The main advantage of this is its ability to optimize the code to be used for parallel execution[42].

[2]Unity DOTS https://unity.com/dots

6.2 Game specifications

The experimental setup which this thesis proposes is an implementation of a shoot em' up game, a sub-genre of a classic point-and-shoot game in which the player must survive as long as possible through endless waves of enemies. The implemented game is inspired by games like "Survivor.io"[26], "Zombero: Archero Hero Shooter"[4], and "Vampire Survivors"[34]. These types of games provide a good base in order to display the advantages of using DOD due to the large number of interacting objects that are often featured in these types of games.

The game which is to be developed will be a first-person shooter in which the objective is to survive through as many waves of enemies as possible. Players will control a single character using the mouse and keyboard in order to navigate the environment. The game features three different weapons, the default handgun with unlimited bullets, together with a shotgun and a machine gun with limited bullets. Players will encounter two types of enemies, a melee-type enemy that attacks the player when within melee range, and a ranged-type enemy that will throw objects at the player when within range. When the player kills an enemy, there is a change of a loot drop containing a random amount of either shotgun or machine gun bullets. Lastly, the game also features a simulated day/night cycle. During the day, players will have to scavenge the environment for loot, while enemies will start appearing and attacking during the night.

6.3 Implementation approach

Game development is an iterative process, victim to many requirement changes. Therefore code flexibility is an important aspect in order to optimize maintainability and development time throughout the process of development. This experimental setup features a simulated development cycle consisting of five iterations for both the OOD and DOD versions. Each iteration will focus on best practices in terms of flexibility and scalability. Flexibility will be accommodated through abstraction and encapsulation in the OOD implementation, while separation of data and behavior ensures a modular flexible design for the DOD implementation. Scalability is ensured through profiling and stress testing of each iteration to identify potential performance-heavy implementations.

To fit the scope of the projects into the limited time frame, advanced features such as animations, particles, sounds, etc, have been left out.

A screenshot of the final prototype can be seen in Figure 6.3.

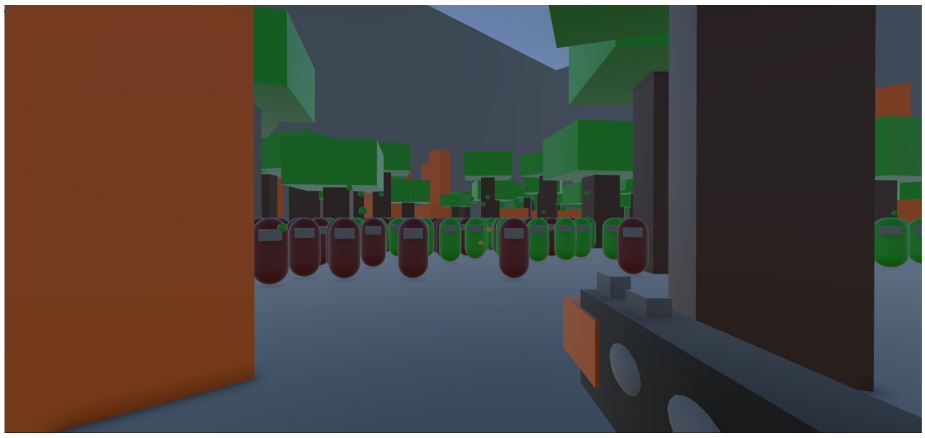

Figure 7: In-game screenshot of the fifth iteration.

6.3.1 Iteration 1: Core Gameplay

The first iteration is focused on implementing the core mechanics of the game. This includes the character controller for movement and shooting, the spawning of enemies and their behaviour, as well as the implementation of an open-world environment.

This iteration focuses on the initial implementation being flexible to prepare for requirement changes in future iterations.

6.3.2 Iteration 2: New Weapons and Ranged Enemies

In the second iteration, two new weapons are added to the game. A shotgun that fires multiple bullets in one shot with a slight spread, and a machine gun featuring a fully automatic mechanic enabling the player to rapidly fire bullets. Furthermore, a new ranged enemy is added that throws objects at the player.

This iteration simulates the requirement change of adding new features to the existing game.

6.3.3 Iteration 3: Day/Night Cycle and Loot System

The third iteration adds a day/night mechanic, which affects the spawning of enemies. Now enemies should be spawned when a new night cycle begins. Furthermore, a looting system is added in which defeated enemies will have a chance to drop ammunition items, that the player can pick up.

The day/night cycle and looting system add more depth to the game, as the player has to manage the ammunition resources in order to use the more powerful shotgun and machine gun weapons for a better chance of surviving the waves of enemies.

This iteration simulates both the addition and modification of already implemented features.

6.3.4 Iteration 4: Enemy Movement and Obstacle Avoidance

The fourth iteration focuses on improving the AI of enemies. Obstacle avoidance is added, so that enemies will navigate through the environment in the game world resulting in more dynamic gameplay.

The iteration consists of both adding a new feature and modifying an existing one.

6.3.5 Iteration 5: Object Pooling

The fifth iteration is focused on optimizing the game's performance using a well-known game programming pattern[31]. Object pooling is implemented to improve the overall performance of the game by minimizing expensive instantiation calls and memory allocation for the ranged enemies when spawning the throwable objects.

The iteration focuses on modifying an already existing feature in order to improve the game's performance.

7 Experimental Setup

This experimental study aimed to validate the flexibility and scalability of the two game versions. To accomplish this, a series of tests and analyses to evaluate both versions under different conditions were conducted.

The tests were conducted independently for each development iteration to evaluate the impact of requirement changes on flexibility and scalability over time. This approach enabled the monitoring of the project's evolution, rather than solely examining the final implementation.

7.1 Flexibility Evaluation

Evaluating the flexibility of the games is essential to ensure the ability to adapt to future changes. To measure code flexibility, several metrics were used to analyze the code for each game version. A detailed code analysis including information about cohesion, coupling, maintainability index, and code change as well as measuring the time spent developing each iteration was performed.

7.1.1 Cohesion

Cohesion and coupling are commonly used metrics used in software engineering to measure the degree to which components of a system work in collaboration to achieve a common goal[7]. In this context cohesion, alongside the other parameters, is an important metric to consider because it can be used to evaluate code flexibility.

In this study, class cohesion was evaluated as a metric to determine the degree to which the methods and attributes of a class are interconnected and aligned

with the overall objectives of the class. High cohesion in a class indicates that its methods and attributes are tightly integrated and function collaboratively towards a shared objective, whereas low cohesion suggests that these components are poorly related and do not adhere to the principle of single responsibility.

As Unity-created games follow a component-based architecture, traditional measures of cohesion such as LCOM (Lack of Cohesion Methods) may not be appropriate[18]. Therefore, a manual code review was conducted for each class to assess the cohesion of the project.

For the DOD version, class cohesion was deemed less relevant since logic and data are separated into components and systems. Instead, the thesis evaluated the interdependence between systems and entities by means of a manual code review. To evaluate the responsibility of each system, the following questions were answered:

1. Is the system responsible for a single, well-defined task?
2. Are all the entities that need to be updated by this system stored in the same data structure?
3. Are there any entities that are stored in these components but don't need to be updated by this system?
4. Are there any calculations or operations that are performed in this system that could be moved to another system to improve cohesion?

7.1.2 Coupling

In software and game development, code coupling plays a significant role in determining the architecture and design of games. The way this concept is approached in OOD and DOD is quite different[7].

In an OOD implementation, the primary focus is on encapsulation in which data and behavior are defined within objects. Objects are designed to represent real-world objects. An example would be a simplified car object, in which data like speed, acceleration, and turn-degree are stored together alongside behaviors like, drive, turn and brake. Behaviors between these objects are typically managed through object references.

OOD promotes loose coupling, which allows for flexibility and modularity in the design, making it easier to reuse and swap out implementations in the case of requirements changes, without affecting other systems.

On the other hand, DOD has a different approach by emphasizing data layout and performance optimizations. The primary focus in a DOD implementation is organizing data in memory to optimize processing and minimize cache misses. Unlike OOD, DOD does not encapsulate behavior within an object. Instead, data and behavior are separated in which behavior is implemented as unique

functions that operate on some specific data. This separation can lead to looser coupling between systems.

For the OOD implementation, code coupling was measured by analyzing the degree to which the code for each game version relied on external dependencies or other components within the project. Additionally, a detailed dependency diagram for each iteration depicting the coupling within the game was added. In the OOD version coupling is defined as a dependency between two classes if a method of one class uses any method or instance variable of the other class[31]. Practically, coupling was measured by analyzing outgoing arrows on the diagrams which depicts direct coupling between classes.

For the DOD implementation, the diagrams and analyzes are slightly different since the data is clearly separated from the behaviour. ECS is designed to avoid coupling between systems. Instead, a series of entity and system has been constructed. This provides an overview of the dependencies different systems has on entities in which they manipulate. Practically, the entity and system dependencies was measured by analyzing outgoing arrows on the diagrams which depicts dependencies from systems to entities.

7.1.3 Maintainability Index

The maintainability index is an index value between 0 and 100 that is calculated to represent the ease of maintaining the code. A high value means improved maintainability[28]. It is calculated as a factor formula that consists of lines of code, cyclomatic complexity and the volume of Halstead [3]. The index is used in several automated code analysis tools, including Microsoft Visual Studio 2019[3], which is used during this project.

7.1.4 Code change and development time

Measuring code change and development time in a project can offer valuable insight into the flexibility of the code base. If changes are made to multiple files throughout the project, it could be an indicator of a lack of flexibility within the code base. Furthermore, the complexity and size of the project can be inferred if developers are required to write extensive lines of code and expend significant amounts of time implementing a requirement change.

To measure these metrics, Microsoft Visual Studio 2019 were utilized to identify lines of code, while Clockify[4] was used to track development time.

[3]Microsoft Visual Studio https://visualstudio.microsoft.com/

[4]Clockify https://clockify.me/lp/home?utm_medium=cpc&utm_source=google&utm_campaign=SCA:BRA_Phrase-Exact:EveryGoal_mCPC:G&utm_agid=137109536779&utm_term=clockify&device=c&gad=1&gclid=CjwKCAjwvJyjBhApEiwAWz2nLbNU6AulrMorOOWD_m5yH1P-IjCJv3VVKI3NK8laVBKCP3-6X9F1URoCIzsQAvD_BwE

7.2 Scalability Evaluation

In order to evaluate the difference in performance focus will be on the computer processor. To test the scalability of the game versions, experiments in which the number of enemies was gradually increased were conducted to measure performance-specific parameters. The frame rate was used as the main performance indicator of elapsed processor time. Additionally, CPU usage time was also measured to provide a more complete picture of the scalability.

An important aspect of measuring the performance of the games is the importance of the used hardware specifications. The results might vary depending on these specifications. A computer with a powerful graphics card and a fast processor will generally be able to run the game more smoothly compared to a system with lower hardware specifications.

To accommodate this, all tests have been performed on two different setups with different hardware specifications. Figure 8 and 9 show the hardware specifications of the two computers used to perform the tests. These specifications were captured with CPU-Z[5].

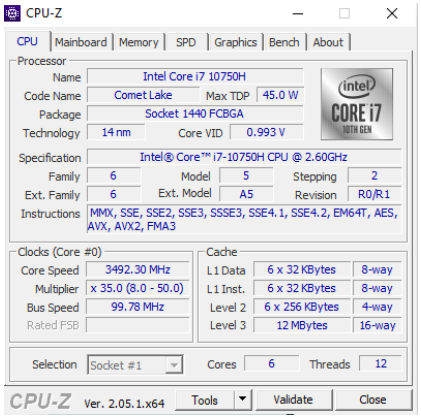

Figure 8: Hardware specifications for PC1.

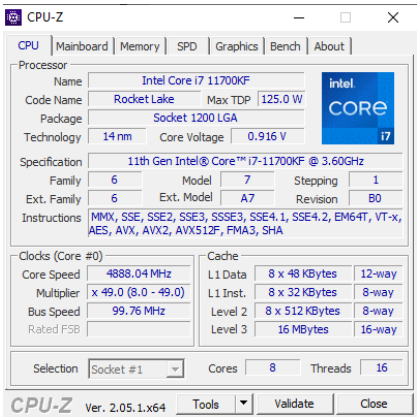

Figure 9: Hardware specifications for PC2.

7.2.1 Frame Rate

Frame rate is a factor that describes the number of consecutive images shown in one second[41]. The Unity Profiler[6] was used to measure the frame rate. In Unity, the frame rate is measured in frames per second (FPS). A higher frame rate means that more frames can be shown every second. Generally, a high frame rate indicates smoother and more responsive gameplay, while a lower frame rate will be experienced as a lagging and unpleasant experience for the players.

[5]CPU-Z https://www.cpuid.com/softwares/cpu-z.html
[6]Unity Profiler https://docs.unity3d.com/Manual/Profiler.html

7.2.2 CPU Usage Time

CPU usage time is the amount of time a process is using the CPU to complete a task[24]. Measuring CPU usage time alongside FPS is important in order to evaluate the performance of a game[16]. While FPS provides a measure of how smoothly the game is executed, CPU usage time provides valuable insights into the overall efficiency and resource utilization. Other factors such as object rendering through the GPU can affect FPS but are not directly related to the implementation[16]. These factors can be excluded by analyzing the CPU usage time. The Unity Profiler was also used to gather this information.

8 Experimentation Results

This section presents the results from the approach introduced in section 7. The results will be presented from each iteration of the two game versions and lastly summarized.

8.1 Cohesion

This section presents the results from the cohesion analysis performed for the OOD and DOD versions.

8.1.1 OOD

This section will describe the cohesion of the individual classes and provide a description of how the cohesion has evolved throughout each iteration. Table 8.1.1 provides an overview of the results from the cohesion analysis of the OOD version of the game (see Appendix **??** for the full analysis).

Class	**1**	**2**	**3**	**4**	**5**
Enemy	3	3	4	4	4
MeleeEnemy	2	1	1	1	1
RangeEnemy	-	1	1	1	2
Throwable	-	1	1	1	1
ThrowablePool	-	-	-	-	1
EnemySpawner	1	1	1	1	2
DayNightController	-	-	2	2	2
LootBehavior	-	-	2	2	2
LootSpawner	-	-	1	1	1
PlayerCamera	1	1	1	1	1
PlayerMovement	1	1	1	1	1
BulletBehavior	3	3	3	3	3
BulletSpawner	1	-	-	-	-
Weapon	-	1	1	1	1
Handgun	-	1	1	1	1
Shotgun	-	1	2	2	2
Machinegun	-	1	2	2	2
WeaponController	-	1	2	2	2

Table 1: Number of responsibilities per class for each iteration of the OOD game version.

Based on the results presented in table 8.1.1 we can see that the first iteration contains a couple of classes that bear multiple responsibilities. Of particular concern with regard to its level of cohesion is the Enemy class, which serves not only as a parent class for the MeleeEnemy class, but also encompasses the responsibilities of enemy movement, attacking, and dying behavior.

The BulletBehavior class, likewise, warrants attention due to its diverse set of responsibilities. It is responsible for updating the bullet position, destruction of the bullet, and reduction of an enemy's health in the event of a collision. Reducing the enemies' health also creates a tight coupling between the BulletBehavior class and the Enemy class since the health data is located in the Enemy class, but modified outside its own encapsulation.

It is worth noting that all remaining classes in the first iteration adhere to the principle of single responsibility. Specifically, the EnemySpawner class is tasked with the responsibility of spawning enemies, the PlayerCamera class is responsible for updating the camera's position and rotation in response to changes in the player's corresponding position and rotation, and the PlayerMovement class is responsible for updating the player's position and rotation in accordance to user input. Lastly, the BulletSpawner class takes on the task of spawning bullets once the player decides to shoot the weapon.

In the second iteration, additional classes were incorporated to integrate the features clarified in Section 6.3.2. A new ranged enemy type was introduced,

which required the reorganization of the MeleeEnemy class that previously handled both the movement and attacking logic. As a result, the Enemy parent class now defines the movement behavior, while the MeleeEnemy and RangeEnemy classes exclusively address distinct attacking behaviors.

Additionally, the BulletSpawner class underwent modifications and now exists as the Weapon class. This class functions as a parent class for multiple weapon types. The individual weapon types inherit the bullet-spawning capability from the Weapon class and hereafter define their specific requirements.

The third iteration of the game has introduced more complex content such as a day/night cycle and a looting system. To support these features, three new classes have been added. The DayNightController class has two responsibilities. Firstly, it updates the day/night cycle, which determines when enemies should spawn. Secondly, it destroys all loot objects when a new night cycle begins.

The LootSpawner class is responsible for spawning loot when the player kills an enemy. This addition required an alteration to the Enemy class, which now includes logic for spawning loot when an enemy is destroyed. The Enemy class directly accesses the loot-spawning method from the LootSpawner class to spawn the loot objects.

The LootBehavior class has two primary responsibilities: updating the rotation of the loot object and updating the ammo amount when the object collides with the player character. To achieve the second task, direct access to the WeaponController class is required, which stores the ammo amount for the shotgun and machine gun. These values are also accessed by the Shotgun and Machinegun classes to decrease the ammo amount when a bullet is fired.

In the fourth iteration, obstacle avoidance was introduced for enemies. This was achieved through a minor modification to the existing movement behavior. Thereby, maintaining the same level of responsibility for all classes in this iteration.

In the fifth iteration, object pooling was implemented for the throwables used by the RangeEnemy class. The ThrowablePool class now handles the spawning of throwable objects. The EnemySpawner script has been modified to include the logic for spawning both enemies and throwable objects. Additionally, the RangeEnemy class now accesses the ThrowablePool to retrieve and restore throwable objects to the pool.

8.1.2 DOD

The following section presents the results of the cohesion analysis conducted on the DOD game version. A summary of the findings can be seen in Table 8.1.2, with the complete analysis available in Appendix **??**.

System	1	2	3	4	5
EnemyBehaviorSystem	3	5	5	5	5
EnemySpawnerSystem	1	1	3	3	4
CharacterControllerSystem	1	1	1	1	1
CameraTargetFollowSystem	1	1	1	1	1
BulletBehaviorSystem	3	3	3	3	3
BulletSpawnSystem	1	1	2	2	2
WeaponControllerSystem	-	1	1	1	1
DayNightSystem	-	-	1	1	1
LootSpawnSystem	-	-	1	1	1
LootBehaviorSystem	-	-	2	2	2

Table 2: Number of responsibilities per system for each implementation iteration of the DOD game version.

Upon analyzing Table 8.1.2, it is evident that the DOD version's initial iteration has one script less than the OOD version. This is due to the absence of inherited classes in the data-oriented design paradigm. A comparison of the implemented systems to the classes from the OOD version reveals that the EnemyBehaviorSystem encompasses all the logic in regard to enemy behavior, including movement, attacking, and destruction.

Similarly, the BulletBehaviorSystem is responsible for updating the bullet's position, reducing the health of the enemies it hits, and eventually destroying the bullet.

It is worth noting that all other systems adhere to the principle of single responsibility.

In the second iteration, the ranged enemy and two new weapon types were introduced. Regarding the new enemy, modifications were made to the EnemyBehaviorSystem, which now encompasses five responsibilities: movement, melee attack, range attack, throwable movement, and enemy destruction.

The two new weapons were integrated into the BulletSpawnSystem, which reuses the original bullet-spawning logic by providing different components to the scheduled job based on the active weapon.

The WeaponControllerSystem is responsible for determining the active weapon by switching between the three different weapons based on user input.

In the third iteration, the day/night cycle and looting systems were introduced, which added complexity to the game. A DayNightSystem was created to update the parameters determining whether it should be day or night in the game logic. As a result, the EnemySpawnerSystem has two new responsibilities besides spawning enemies. It is now responsible for accessing the DayNightNightSystem directly to execute the parameter-update job and for destroying all loot entities when a day cycle ends by accessing the LootBehaviorSystem directly.

Two new systems were also introduced: the LootSpawnSystem, responsible

for spawning loot, and the LootBehaviorSystem, responsible for rotating the loot entities and updating the ammo amount when colliding with the player character.

Unlike the OOD version, loot spawning is not handled within the enemy destruction logic. Instead, a flag approach was implemented, enabling a specific component for the enemy entities marking them as dead. This new archetype is used to spawn loot at the position of enemies marked as dead before the enemy destruction job is executed.

The ammunition logic also differs from the OOD version, with a single ammo archetype created that contains data for both shotgun and machine gun ammunition values. This archetype is used in the BulletSpawnSystem to determine whether enough bullets are available before spawning bullets and to decrease the correct ammunition value when a bullet is fired.

As in the OOD version, the fourth iteration only required a minor modification to the EnemyBehaviorSystem to incorporate obstacle avoidance into the movement behavior. Thereby, we managed to maintain the same level of responsibility for all systems.

In the fifth iteration of the project, object pooling was introduced as a means of managing enemy throwables. Specifically, this was achieved by augmenting the existing throwable entity archetype with new component data that indicates whether the entity is currently in use by an enemy or is available for use. Modifications were then made to the EnemyBehaviorSystem such that the range attack behavior is now triggered only when a throwable is marked as available. Upon collision, the throwable is simply marked as available once again.

Additionally, the EnemySpawnSystem was enhanced to instantiate a specified number of throwables in accordance with the number of ranged enemies. As a result of these updates, the EnemySpawnSystem now performs four distinct responsibilities: spawning enemies, updating day/night parameters, destroying loot, and spawning throwables.

8.2 Coupling

This section presents the result from the performed coupling analysis of the two game versions.

8.2.1 OOD

This section presents the results from the approach introduced in Section 7 for the OOD implementation. An in-depth analysis of the constructed dependency diagrams which are presented in this section will be performed. The diagrams show dependencies within the project in which straight arrow lines depict a direct dependency from the class. The arrow starts from the class that is dependent on the other class in which the arrow points at. The dotted lines show inheritance, in which the class from which the arrow is starting inherits from the

class in which the arrow is pointing. The classes have also been categorized in colour-coded boxes to illustrate their responsibilities in terms of different game features. These categories are:

- Blue boxes are all related to weapons.
- Orange boxes are all related to the character's movement.
- Green boxes are all related to enemies.
- Purple boxes are all related to the day/night cycle.
- Yellow boxes are all related to the loot objects.

Even though some of the classes depend on the same game objects, like the PlayerCamera and PlayerMovement which both depend on the player character game object, these kinds of dependencies were not included in the analysis. Instead, the focus was on exterminated code-coupling between classes.

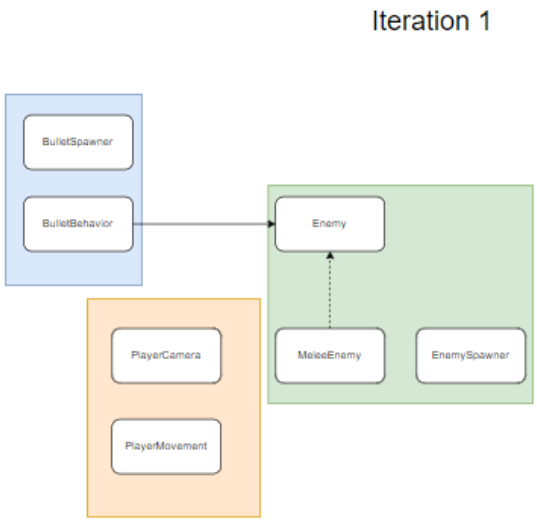

Figure 10: Dependency diagram for the first iteration of the OOD implementation.

Iteration 1: Based on the diagram presented in Figure 8.2.1 we can see that the overall coupling of the classes is fairly low for the first iteration. We have a single parent class, the Enemy class, from which the MeleeEnemy inherits. Furthermore, we see a coupling between the BulletBehavior class to the Enemy

class. This is caused by a direct coupling to the MeleeEnemy class attached to an enemy object that the BulletBehavior class directly accesses in order to change the health parameter on collision with a bullet.

Overall, the first iteration is fairly decoupled only persisting one tight coupling between two classes.

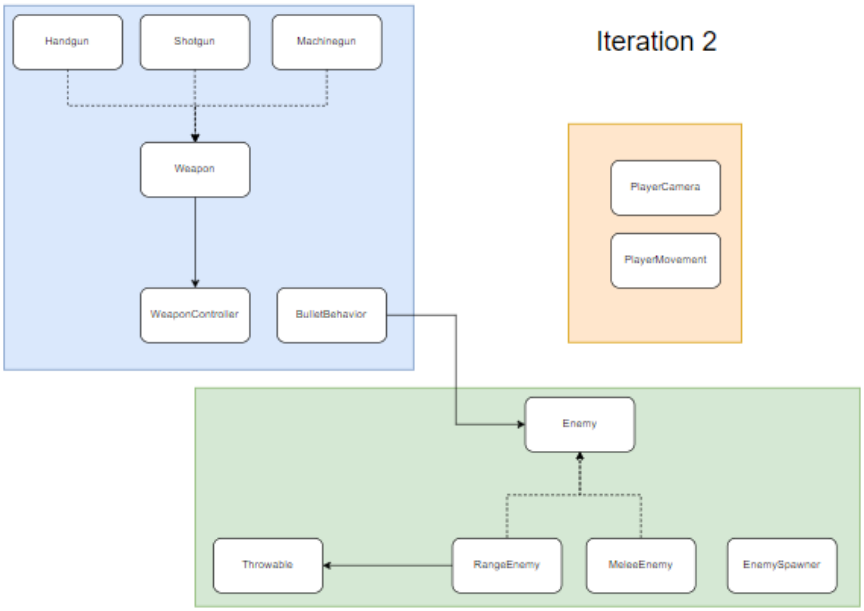

Figure 11: Dependency diagram for the first iteration of the OOD implementation.

Iteation 2: Figure 8.2.1 shows the dependencies for the second iteration. This iteration introduced two new weapon types and a new enemy type. We see that the BulletSpawner class from the first iteration was refactored into a parent class named Weapon. The three derived classes, Handgun, Shotgun, and Machinegun overwrites the bullet spawning method to apply specific spawning logic based on the weapon type. We also see that the Weapon class has a dependency on the introduced WeaponController class that determines which weapon should be used based on user input.

Furthermore, this iteration also introduced the new RangeEnemy class which inherits from the Enemy class that defines movement, attacking and destruction behavior for all enemy types. The RangeEnemy class depends on the Throwable class that holds the logic for the throwable object. The dependency is caused by the throwable spawning logic encapsulated in the RangeEnemy class. A new throwable gameobject is being instantiated in which we directly add the Throwable class and sets parameters for the enemy position and the player

position. These values are used to update the position of the throwable object.

Overall, the iteration still persists with fairly low coupling between classes. The most concerning one is still the dependency between the BulletBehavior and Enemy class introduced in the first iteration.

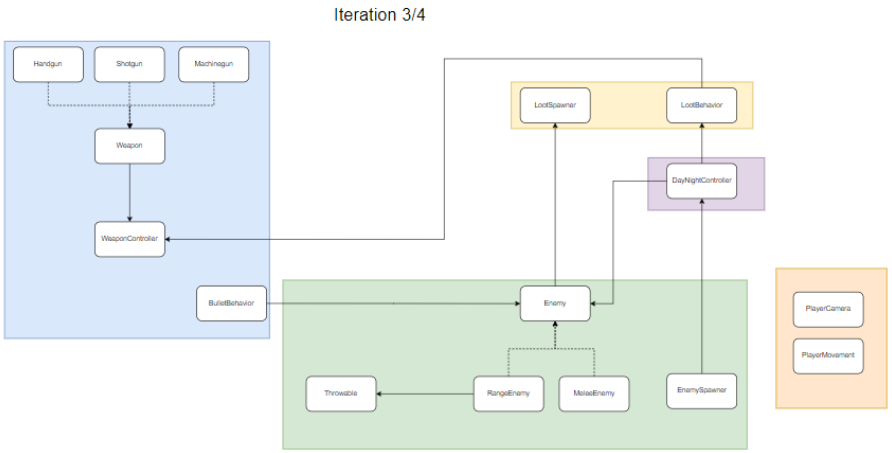

Figure 12: Dependency diagram for the first iteration of the OOD implementation.

Iteration 3: Figure **??** shows the dependencies of the third and fourth iterations. The third iteration introduced two new features to the game. The DayNightController class contains the logic for the day/night cycle determining when enemies should be spawned. We see that this significantly increased the coupling within the codebase. This new class has a dependency on the Enemy class caused by the need to find all active game objects that contain the Enemy class in order to determine if any enemies are still alive. This coupling is fairly loose since the class do not access any methods or variables from the class. Therefore, it can be determined as a reasonable coupling in order to implement the needed functionality in this class.

Furthermore, the DayNightController is also depended on the introduced LootBehavior class. This dependency is caused by the need for querying all game objects that have the LootBehavior class attached. A list of all game objects that contains the class will be constructed in order to destroy all these game objects, once a new night cycle is about to start. Like the previous case with the Enemy class, this is a loose coupling, but there is a higher risk of affecting other systems since we destroy the game objects.

The introduction of the day/night cycle also introduced a coupling between the EnemySpawner and the DayNightController classes. This arises because the EnemySpawner needs to access the variables from the DayNightController class that determines if enemies should be spawned. Once enemies have been

spawned, we directly set the parameters that state if enemies have been spawned stored in the DayNightController from the EnemySpawner. This is a fairly tight coupling.

Furthermore, the iteration also introduced the looting system that enables the player to gather ammunition for the shotgun and machine gun. This feature required the implementation of the LootBehavior class as mentioned above. The class is responsible for updating the ammunition amount when the player collides with a loot object. This requires a dependency on the WeaponController class that contains ammunition data. The LootBehavior class directly manipulates this data which creates a tight coupling between these two classes.

Additionally, the LootSpawn class responsible for spawning loot once an enemy dies was implemented. The Enemy class directly calls the method for spawning loot before destroying the enemy gameobject.

We see that the overall coupling significantly increased with this iteration having several tight couplings between the classes.

Iteration 4: The fourth iteration introduced obstacle avoidance which only required a small change to the Enemy class. Therefore, the overall coupling for the project maintains the same as for the third iteration.

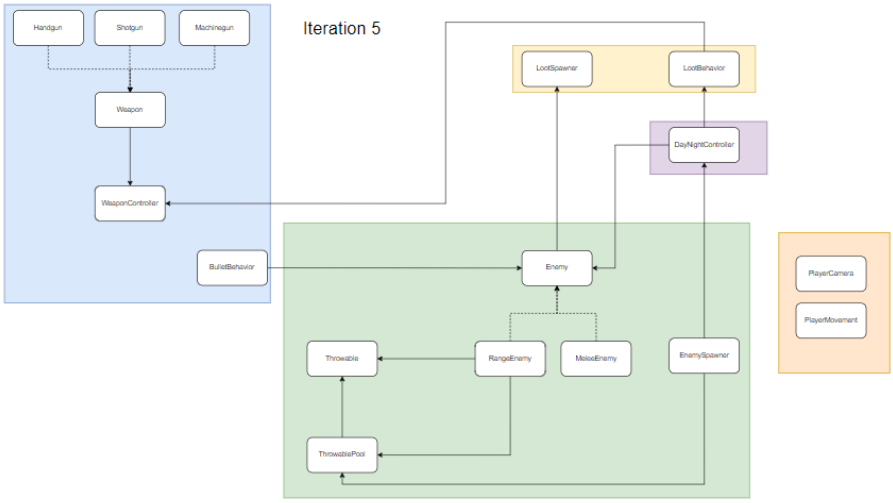

Figure 13: Dependency diagram for the first iteration of the OOD implementation.

Iteration 5: In the last iteration throwable object pooling was introduced. This resulted in the implementation of the ThrowablePool class that is responsible for instantiating a certain amount of throwable objects. Furthermore, the

class defines logic for the ranged enemies to access in order to use inactive throwables. This introduced three new dependencies. The ThrowablePool class is dependent on the Throwable class which is caused by the need for instantiating throwable objects and setting game objects as either active or inactive when needed. This is a fairly loose coupling because we do not directly use any of the methods or parameters from the class but instead, query game objects with the attached class.

Furthermore, the RangeEnemy class has a coupling with the ThrowablePool class since we directly access and call the methods in order to manage the created object pool.

Lastly, the EnemySpawner class has a dependency on the ThrowablePool. This is caused by the EnemySpawner calling the method for creating the object pool and hence becoming responsible for instantiating the throwable objects.

Overall, the coupling of the game has increased with this iteration resulting in the final implementation having fairly tight coupling between the classes.

8.2.2 DOD

This section presents the findings obtained from the coupling analysis conducted on the implementation of the DOD version as introduced in Section 7. To depict the relationships between systems and their respective archetypes involved in data access and transformation, a series of dependency diagrams have been constructed. Notably, the approach employed by DOD deviates slightly from OOD, as it places a strong emphasis on data and its manipulation. The data-first approach reduces the need for complex object hierarchies and minimizes dependencies of systems. Therefore, the system coupling will naturally be reduced since systems are designed to be executed independently of each other. Furthermore, the necessity of accessing objects to retrieve data information has been removed with this approach.

The diagrams depict the archetypes as squares, each with their components listed underneath. Some of the archetypes have a gradient purple colour illustrating that only a single entity exists of this archetype. Systems are depicted as squares with rounded corners. Each system has a color depicting a responsibility category. Systems are categorized into the following:

- Green systems are all player character manipulating, responsible for controlling the player character.
- Purple systems are all enemy manipulating, responsible for spawning and controlling the enemy behavior.
- Light blue systems are all bullet manipulating, responsible for spawning and controlling the behavior of weapons.
- Yellow systems are manipulating the day/night cycle.

- Orange systems are all loot manipulating, responsible for spawning and controlling the behavior of loot.

Each archetype dependency is depicted with an arrow going from a system to an archetype. Some of these arrows have a small text saying "create" meaning that this system is responsible for creating entities of that archetype.

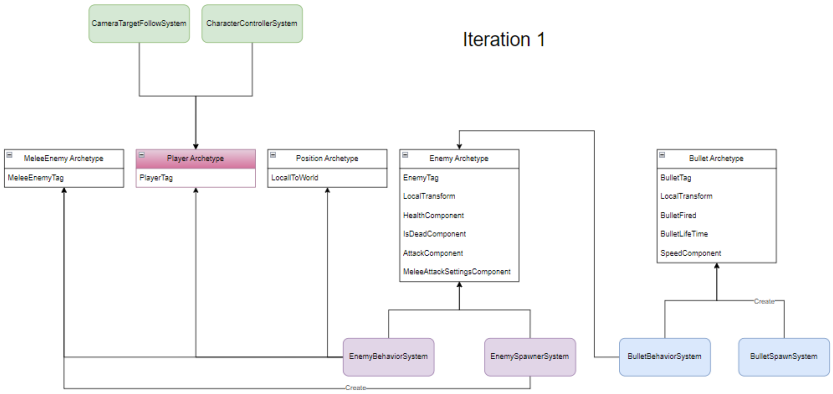

Figure 14: Dependency diagram for the first iteration of the DOD implementation.

Iteration 1: From the diagram presented in Figure 8.2.2 we can observe that the first iteration consists of 5 archetypes and 6 systems. The two systems CameraTargetFollowSystem and CharacterControllerSystem manipulate the player character archetype. The player archetype is furthermore accessed by the EnemyBehaviorSystem to check for collisions and move the enemy toward the player.

The EnemySpawnerSystem and EnemyBehaviorSystem work in conjunction to control the behavior of enemies. The EnemySpawnerSystem will create entities representing an enemy archetype, that is used within the EnemyBehaviorSystem to control the behavior of enemies.

The BulletSpawnSystem and BulletBehaviorSystem too work in conjunction to control the behavior of bullets fired by the player. Once bullets have been spawned, the BulletBehaviorSystem is responsible for manipulating the component data of bullet archetypes.

Overall we observe a completely decoupled codebase where each system works independently.

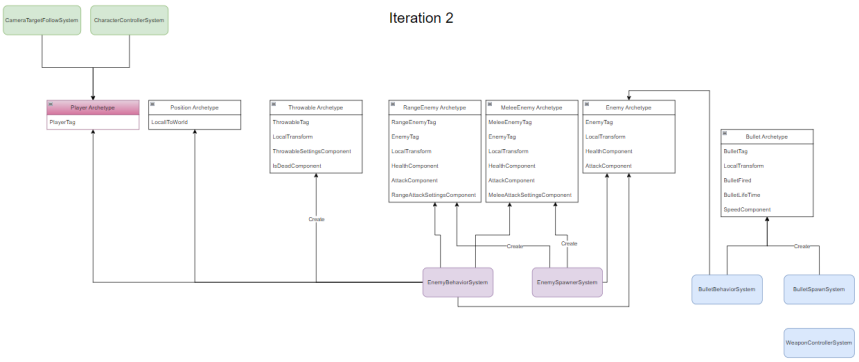

Figure 15: Dependency diagram for the second iteration of the DOD implementation.

Iteration 2: From the second iteration shown in Figure 8.2.2 we can see that two new archetypes have been added while the MeleeEnemy archetype has been modified. Now both the EnemySpawnerSystem and EnemyBehaviorSystem have a dependency on the MeleeEnemy and RangeEnemy archetypes, while the EnemyBehaviorSystem is also responsible for creating and manipulating the throwable entities.

A new system, WeaponControllerSystem has also been added in this iteration. This system can not be identified as a data-oriented system since it inherits from the MonoBehavior class used for object-oriented programming. The WeaponControllerSystem is a standalone system responsible for changing the weapon gameobject created outside the ECS implementation. This is introduced as a small hack since weapon objects are attached to the camera, that can not be converted and used by the ECS.

Overall the fully decoupled implementation persists in this iteration.

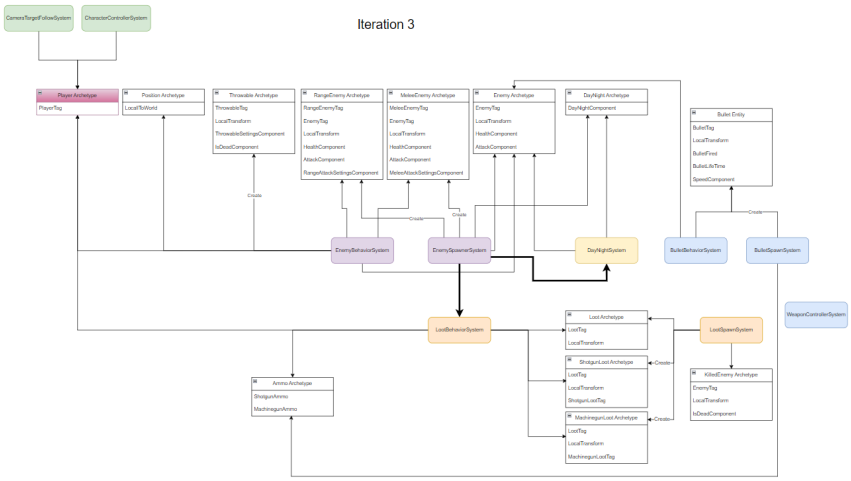

Figure 16: Dependency diagram for the third iteration of the DOD implementation.

Iteration 3: The third iteration introduced the day/night cycle and looting system. From Figure 8.2.2 we see that 6 new archetypes have been added alongside 3 new systems. The LootSpawnSystem is responsible for spawning loot at the position of killed enemies. The data of the spawned loot is manipulated by the LootBehaviorSystem which is responsible for updating the new ammunition archetype when the player character collides with a loot entity.

Also, the DayNightSystem has been introduced in this iteration that simply updates the DayNight archetype that determines when enemies should be spawned and loot should be destroyed.

We see that the EnemySpawnerSystem has two new dependencies on the DayNightSystem and the LootBehaviorSystem. This is caused by the need for directly accessing and executing jobs defined in the two systems. These dependencies are marked in Figure 8.2.2 with a bold line. This can be identified as a tight coupling because the EnemySpawnerSystem is directly dependent on some functionality of other systems.

Overall, the coupling of the implementation has increased because a system now directly calls functions defined in other systems.

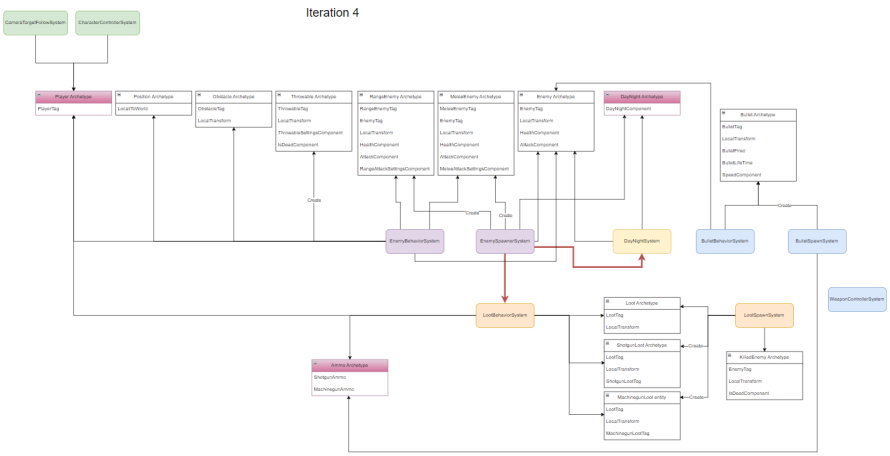

Figure 17: Dependency diagram for the fourth iteration of the DOD implementation.

Iteration 4: The fourth iteration only required a small change in order to implement obstacle avoidance. Figure 8.2.2 shows that the obstacle archetype has been added which the EnemyBehaviorSystem uses to check for colliding obstacles.

Overall, the level of coupling persists from the previous iteration.

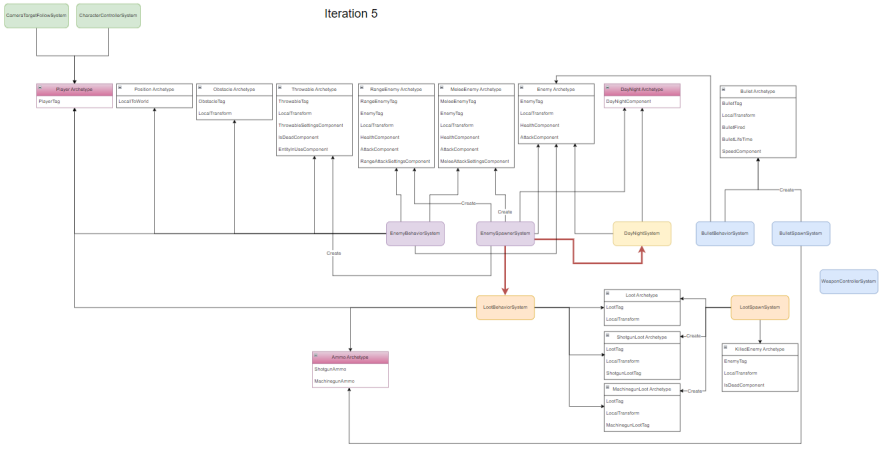

Figure 18: Dependency diagram for the fifth iteration of the DOD implementation.

Iteration 5: The fifth iteration also featured a fairly small change to the implementation. In Figure 8.2.2 we see that the entities in the Throwable archetype

have been modified with a new component, EntityInUseComponent. This component is a special type that can be set as visible or invisible when querying for Throwable archetypes. Instead of having the spawning behavior in the EnemyBehaviorSystem it was refactored to the EnemySpawnSystem, which will spawn a certain amount of throwable entities with this new component. The EnemyBehaviorSystem simply set the new component to visible once a throwable needs to be used by an enemy, indicating that this is in use and cannot be used by any other enemies. Once the throwable is no longer in use, the component is set to invisible again, indicating that it can be used by any enemy that needs to access it.

Overall, the coupling level persists from the previous iteration.

8.3 Maintainability index

This section presents the results from the maintainability index calculations performed for both the OOD and DOD versions. A comparison in the average development of the index can be seen in Figure 19. The following two subsections will go into depth with the development of the specific classes and systems.

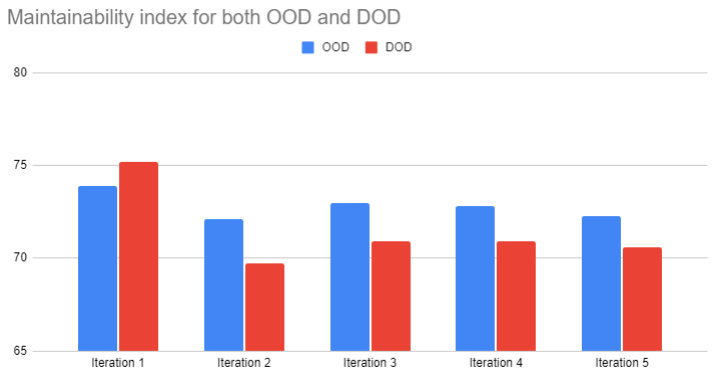

Figure 19: Development of the average maintainability index value throughout all iteration for both the OOD and DOD version. [22].

8.3.1 OOD

In Table 8.3.1, the maintainability index for each class can be seen. It can be observed that the BulletSpawner and Weapon have high maintainability scores in the first and second iterations while the weapon script consistently maintains high throughout all iterations. Also, the Enemy class has a high maintainability index for the first iteration but decreases as more complex behavior were refactored in the second iteration.

Other classes, such as the BulletBehavior, MeleeEnemy, and PlayerCamera have

relatively consistent maintainability scores throughout all iterations. The EnemySpawner class has high scores in the first iteration but gradually decreases in the subsequent iterations due to the increased level of responsibility and cyclomatic complexity that comes when adding logic for spawning range enemies and managing the object pool in the fifth iteration.

New classes introduces in the third iteration, such as the DayNightController and LootSpawner scripts, have lower maintainability scores, which can be attributed to their complexity.

Class	**1**	**2**	**3**	**4**	**5**
Enemy	81	75	75	71	71
MeleeEnemy	76	70	70	70	70
RangeEnemy	-	69	69	69	63
Throwable	-	61	71	71	71
ThrowablePool	-	-	-	-	74
EnemySpawner	79	70	70	70	67
DayNightController	-	-	66	68	68
LootBehavior	-	-	76	76	76
LootSpawner	-	-	74	74	74
PlayerCamera	75	75	75	75	75
PlayerMovement	60	60	60	60	60
BulletBehavior	72	71	71	71	71
BulletSpawner	90	-	-	-	-
Weapon	-	91	91	91	91
Handgun	-	82	82	82	82
Shotgun	-	76	72	72	72
Machinegun	-	76	75	75	75
WeaponController	-	67	67	67	67

Table 3: Maintainability index per class for each implementation iteration for the OOD version.

8.3.2 DOD

In Table 8.3.2 the maintainability index for each iteration of the implemented systems can be seen. For the specific systems, it can be observed that the EnemySpawnerSystem and BulletSpawnSystem have higher maintainability index scores during the first iteration but gradually decrease in the subsequent iterations. On the other hand, the EnemyBehaviorSystem, BulletBehaviorSystem, and WeaponControllerSystem have consistent scores throughout all iterations.

The second iteration which introduced the ranged enemy clearly affects the maintainability index for the EnemyBehaviorSystem since it got two new responsibilities which added increased cyclomatic complexity to the existing system.

The same explanation can be reasoned for the change in the BulletSpawnSystem, which now contains more complex behavior in order to determine which weapon to spawn bullets for.

In the third iteration, two new features were added to the game, encapsulated as the DayNightSystem, LootSpawnSystem, and LootBehaviorSystem. These systems have a moderate level of maintainability, which can be addressed to their complexity.

While the fourth iteration did not have an impact on the maintainability index, the fifth iteration introduced object pooling. We see that this directly affects both the EnemyBehaviorSystem and the EnemySpawnerSystem since more complex logic is added to these systems.

System	**1**	**2**	**3**	**4**	**5**
EnemyBehaviorSystem	70	64	64	64	61
EnemySpawnerSystem	81	81	71	71	66
CharacterControllerSystem	77	77	77	77	77
CameraTargetFollowSystem	80	80	80	80	80
BulletBehaviorSystem	68	68	68	68	68
BulletSpawnSystem	81	63	63	63	63
WeaponControllerSystem	-	66	66	66	66
DayNightSystem	-	-	71	71	71
LootSpawnSystem	-	-	75	75	75
LootBehaviorSystem	-	-	74	74	74

Table 4: Maintainability index per system for each implementation iteration for the DOD version.

8.4 Code change and development time

This study analyzed the changes made to the source code of the two game versions. Table 8.4 provides an overview of the total lines of code in each version for all iterations. See Appendix **??** and **??** for a fully detailed analysis including changes to each class and system. The development time for each iteration was also measured. The results can be seen in Table 8.4.

Game version	**1**	**2**	**3**	**4**	**5**
OOD	229	422	574	583	661
DOD	791	1129	1551	1608	1671

Table 5: Total lines of code in both game versions for each iteration.

The total lines of code represent the overall size and complexity of the game versions. Inspecting the table, we can see that the DOD version is consistently larger than the OOD version across all iterations. We see that the DOD version

is about three times larger in comparison to the OOD version throughout the whole project.

Game version	1	2	3	4	5
OOD	11,2	6,8	9,3	4,2	7,6
DOD	17,4	11,1	11,6	3,9	7,1

Table 6: Hours spent on each iteration for both game versions.

Table 8.4 shows the overall development time for each iteration. We can see that the DOD version takes significantly longer during the initial setup which required a lot more setup in order to get the application to work as intended.

Also for the second and third iterations, the DOD version required more development time, while in the fourth and fifth iterations, the development time was surprisingly faster than the OOD version.

In total, the OOD version took 39,1 hours to develop, while the DOD version required 51,1 working hours to finish which is 30,67% more than the OOD version.

8.5 Scalability

This section presents the results from the scalability analysis performed on the two game versions.

8.5.1 Frame Rate

Figures 20 and 21 show the average FPS for each iteration tested with a different number of enemies for the OOD version on the two computers.

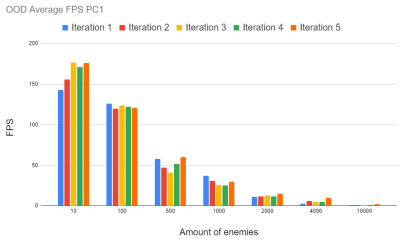

Figure 20: Average FPS OOD overview performed on PC1.

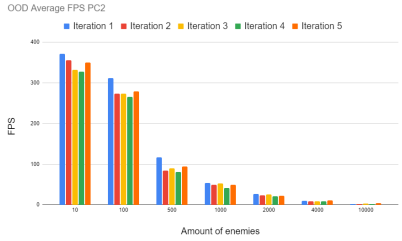

Figure 21: Average FPS OOD overview performed on PC2.

From the results, we see that there is a general FPS drop when more enemies are spawned. Generally, the first iteration has a higher FPS throughout all iterations because no complex behavior was present for this iteration. As more complex behavior was added throughout the iterations, we see that the average

FPS drops slightly but not significantly enough for the players to recognize when playing.

It can also be observed that the fifth iteration slightly increased the average FPS as object pooling was implemented.

Furthermore, the results also showed that PC2 generally had higher average FPS values throughout all iterations when comparing the two figures.

Figures 22 and 23 show the average FPS for each iteration of the DOD implementation performed on the two computers.

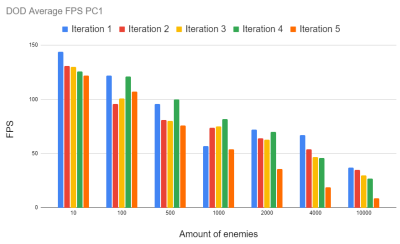

Figure 22: Average FPS DOD overview performed on PC1.

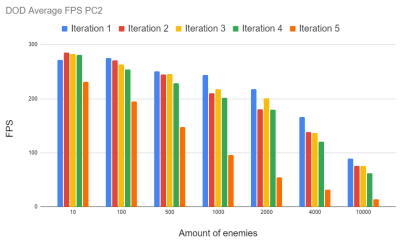

Figure 23: Average FPS DOD overview performed on PC2.

The results show that the DOD generally has a higher average FPS compared to the OOD implementation. Generally, an average of 60 FPS is accepted by developers as an acceptable value for games[36]. From the results, it can be observed that the OOD implementation quickly drops below 60 FPS when 500 enemies are spawned. For the DOD implementation, most iterations drop below 60 FPS when 4000 enemies are spawned.

Inspecting the individual iterations we see that the average FPS slightly decreases throughout iterations 2, 3, and 4, while it decreases drastically for the fifth iteration that introduced object pooling. The average FPS for the fifth iteration drop below 60 FPS already at 1000 spawned enemies for PC1.

8.5.2 CPU Usage Time

This section presents the results from the conducted CPU usage time analysis. During this, the CPU usage time of the player loop was measured for each iteration with a varying number of enemies.

Table 8.5.2 shows the results from the conducted tests of the OOD version. From this, it can be observed that the CPU usage time quickly accelerates with the number of enemies increasing for all iterations. It can also be observed that the CPU usage time is somewhat consistent through all iterations independent of the number of enemies. For example, in the case with 10 spawned enemies, the average CPU usage time ranges from 1,1ms to 2,82ms, and for the case with

1000 enemies, it ranges from 17,26ms to 27,56 ms. This indicates that the CPU has to spend more time when complex logic is added to the game.

Furthermore, we can also observe that the results from PC2 are generally better compared to PC1 through all cases.

Table 8.5.2 shows the results from the conducted tests of the DOD version. It can be observed that the CPU usage time slowly increases when more enemies are spawned throughout all iterations. The difference between the values for each iteration is fairly low for iterations 1, 2, 3, and 4 while the fifth iteration increases significantly faster.

Furthermore, it can also be observed that PC2 performs better throughout all tests in comparison to PC1.

OOD PC1	**10**	**100**	**500**	**1000**	**2000**	**4000**	**10000**
Iteration 1	1,88	3,93	15,9	17,26	46,61	120,2	301,4
Iteration 2	2,82	2,55	13,81	22	47,11	106,46	316,84
Iteration 3	1,17	1,47	11,8	25,15	50,56	97,73	258,02
Iteration 4	1,16	3,22	12,67	27	67,85	133,23	514,66
Iteration 5	1,1	2,28	13,42	27,56	73,05	116,86	462,85
OOD PC2	**10**	**100**	**500**	**1000**	**2000**	**4000**	**10000**
Iteration 1	0,65	0,87	4,69	10,66	27,38	82,96	215,5
Iteration 2	0,58	1,13	6,48	13,91	31,28	90,71	277,71
Iteration 3	0,67	1,11	6,82	14,34	32,55	82,99	217,68
Iteration 4	0,57	1,24	6,97	17,49	40,19	97,42	205,01
Iteration 5	0,7	1,13	6,76	15,17	38,78	82,65	197,12

Table 7: OOD CPU usage time in milliseconds for each iteration with various numbers of spawned enemies for both computers.

DOD PC1	**10**	**100**	**500**	**1000**	**2000**	**4000**	**10000**
Iteration 1	2,12	2,6	5,07	6,28	6,53	9,85	15,77
Iteration 2	2,08	1,69	3,22	5,51	8,01	11,51	15,97
Iteration 3	2,11	2,16	5,25	6,79	8,45	13,14	30,29
Iteration 4	2,52	2,53	3,82	5,94	9,8	16,94	26,58
Iteration 5	2,22	3,52	7,41	12,47	21,57	43,24	102,65
DOD PC2	**10**	**100**	**500**	**1000**	**2000**	**4000**	**10000**
Iteration 1	0,97	0,93	1,45	2,26	2,35	3,27	7,76
Iteration 2	0,91	1,13	1,52	2,56	2,23	3,72	8,8
Iteration 3	1,17	1,37	1,89	2,28	2,32	3,93	8,83
Iteration 4	1,18	1,16	1,57	2,59	2,68	5,25	12,33
Iteration 5	1,09	1,82	3,45	6,78	13,11	29,12	68,46

Table 8: DOD CPU usage time in milliseconds for each iteration with various numbers of spawned enemies for both computers.

9 Discussion

The previous section presented the results from the conducted flexibility and scalability analysis of the two game versions. This section will discuss the findings including the effectiveness and limitations of the research.

9.1 Flexibility

From the results presented in Section 8 information on different metrics was gathered in order to validate the flexibility of the two game versions.

9.1.1 Cohesion

The cohesion analysis showed a fairly high level of cohesion for both versions, but at the same time identified some possible issues with the implementations.

OOD
For the OOD version especially the Enemy class proved to develop some issues in terms of cohesion throughout the iterations. At first, this class acted as a parent class for the two enemy types which they inherit to override behaviors defined in the Enemy class. The class defined responsibilities for movement, attacking, and destruction behavior. In an ideal situation, this should be refactored into three different component classes to improve the flexibility of the project. Imagine that we later in the development would like to implement a scavenging enemy, whose only task was to search for loot that it could steal. This new enemy type would still have attacking behavior when inheriting from the Enemy class, but it would never be used. Or imagine a defensive tower enemy, that shoots projectiles at the player character. This enemy would not need any movement-defined behaviors.

DOD
Likewise for the EnemyBehaviorSystem in the DOD implementation. This system was responsible for defining all behavior of both the melee and ranged enemies, as well as defining the behavior for the throwable spawned by the ranged enemies. As a result of this, the system ended up being fair complex and difficult to maintain. Optimally, we should refactor this into three different systems, one for defining the movement of enemies, another for defining attacking behavior, and a separate system for the logic defining the behaviour of throwable entities to make the system less complex and overwhelming.

Cohesion in game-specific code tends to be a fluffy concept. One reason for this is that there is no universally accepted definition of cohesion in game development[14]. There are some general principles that can be applied, such as the principle of single responsibility and the urge to avoid tight coupling between classes. Often, developers lean towards encapsulating all behavior defining the logic of gameobjects in the same class, as was the case for the Enemy class in

the OOD version. This proved to be both a good and bad approach, since the behavior of an enemy was clearly defined in one class, but would be a problem for the above-described scenario. This is where a component-based approach, such as the ECS approach used here, is a nice fit for game development. ECS emphasizes the *Composition over Inheritance* principle[9]. Instead of defining behavior through class hierarchies, ECS promotes composition of small reusable data components. This is employed to create data structures that are optimized for cache efficiency. Developers can create flexible and modular data structures, that can be combined in different ways to achieve the desired logic of systems[27].

ECS forces the developers to have a fairly flat code structure with no complex class hierarchies. Therefore, developers can easily fall into the trap of defining all logic in the same system when experimenting and implementing new features. It is important when working with this paradigm, that developers try to stay organized by separating the responsibilities of entities into appropriate systems to make the code more readable and maintainable.

Overall, the DOD implementation proved to be slightly less cohesive in comparison to the OOD implementation. This is mainly caused by the missing inheritance in the enemy behavior defined in the EnemyBehaviorSystem. While the OOD implementation utilized inheritance to define two separate classes for the enemies, all enemy behavior was defined in the EnemyBehaviorSystem in the DOD version.

9.1.2 Coupling

OOD

Based on the results from the coupling analysis of the OOD implementation we can see that the coupling changes throughout the iterations. There is evidence that the overall coupling of the game increased over time. While the initial design had relatively low coupling, the introduction of new features and dependencies led to tighter coupling between the classes for the OOD implementation.

In the first iteration, the overall coupling of the classes can be considered fairly low, with only one tight coupling between the BulletBehavior and Enemy classes. This indicates that the initial design had relatively good decoupling. However, when more features were added to the game, as for the second iteration that introduced new weapons and enemies, the coupling between classes only increased throughout the development process.

In terms of coupling, the third iteration was the most crucial since it introduced features that directly affected features of other classes. These additions significantly increased the coupling within the codebase because we directly had to retrieve information from the DayNightController and the LootBehavior classes to determine when enemies should be spawned and loot destroyed.

To mitigate the negative effects of coupling, it is essential to apply good software design principles for example encapsulation. This has been applied to some extent, but some classes eventually ended up having multiple responsibilities as

explained in Section 8. To promote loose coupling other design patterns could have been utilized. For example, the observer pattern could be used to notify classes, once a change has happened in one class that triggers some behavior in another class. Like the example with the EnemySpawner class that should spawn enemies once a new night cycle started. The observer pattern has two main components: the subject and the observer[31]. In our case the DayNightController would act as the subject and the EnemySpawner class would act as the observer. The observer would subscribe to the subject and receive a notification whenever the day/night cycle state changes. This way, the need to directly access the class variables and functionality in order to spawn enemies could be eliminated. The subject does not need to know the specific details of the observer. All classes are treated as generic objects allowing for multiple observers to subscribe to the subject. This means that new observers can be added and existing ones can be modified without affecting the subject. They are purely connected through a common interface, allowing for greater flexibility of the classes.

DOD

The coupling analysis for the DOD implementation was slightly different compared to the OOD implementation because DOD promotes the separation of data and transformation. The need for class hierarchies and encapsulation has been removed with this approach promoting more decoupled systems made to be run independently of each other's existence.

The DOD implementation emphasizes data manipulation over complex class hierarchies. This approach reduces dependencies between systems and promotes loose coupling. By focusing on data and its transformation, the need for accessing classes to retrieve information is eliminated.

In the initial iteration, all systems work independently from each other. The coupling between systems and archetypes is relatively low, although it can be argued that the EnemyBehaviorSystem and the BulletBehaviorSystem are dependent on the work of the EnemySpawnerSystem and BulletSpawnerSystem to perform their work. This coupling is fairly low since the behavior systems will just idle until entities of the corresponding archetype are available for data manipulation.

The coupling continues to be low throughout the second iteration as the responsibility of the EnemyBehaviorSystem increases and the data and transformations are still separated.

The third iteration introduces some problems in terms of coupling. This iteration implemented the DayNightSystem that holds the logic for updating the day/night cycle. One problem is, that the system has a job that is directly executed by the EnemySpawnerSystem which creates a tight coupling between the two systems. In reality, this job is only defined in the DayNightSystem but is never executed within this system. This means that the job could have been defined anywhere else in the implementation without affecting the functionality. To eliminate this coupling, we could simply refactor the job to the EnemySpawnerSystem since it is only used in that system. Optimally, the logic

should be refactored to respective systems in order to follow the single responsibility principle and decrease coupling. However, one could argue that this type of coupling does not adhere to the definition of coupling in OOD. As previously mentioned, coupling refers to a condition where a class relies on the data or functionality of another class. In the context of DOD, components are structured around data and its transformations, rather than encapsulated objects with associated behavior[19]. DOD systems are designed to operate on specific components, thereby reducing coupling between systems to data dependencies rather than object dependencies.

The fourth and fifth iterations only required minor changes in relation to obstacle avoidance and throwable object pooling due to the component-based design facilitated in the DOD implementation.

Overall, the DOD implementation demonstrates a reduction in coupling compared to the OOD implementation. The emphasis on data manipulation and the ability to design systems that execute independently contribute to loose coupling between systems. However, the third iteration introduced some tight coupling between systems which should be avoided to increase the overall flexibility. It is crucial to strike a balance between the benefits of DOD and the potential risks associated with coupling between systems if not developed carefully. Regular assessments and refactoring can help optimize the coupling and maintain the overall structure of the game.

9.1.3 Maintainability index

The maintainability index analysis provides valuable insights into the maintainability of different classes and systems across multiple iterations. When inspecting the overall analysis of the OOD implementation we can see that the Weapon class maintained a high level of maintainability throughout all iterations which indicates that the class is well-designed and easy to maintain. The high score suggests that it has clear and cohesive responsibilities, making it less prone to issues if modifications are applied.

OOD

The Enemy class which was modified in every iteration starts with a high maintainability index in the first iteration but gradually decreases as more complex behaviors are introduced. The decrease suggests that the introduction of complex behavior has made the class harder to maintain. As the cohesion analysis also showed, the complexity rose because more features were added to the class increasing both the amount of code and the responsibilities making it more difficult to maintain the class. It highlights the need for careful design and refactoring to manage increasing complexity to keep it at a maintainable level.

Also, the EnemySpawner class starts with a high maintainability score in the first iteration but gradually decreases in subsequent iterations. This is caused by the increased responsibility and cyclomatic complexity, particularly in the fifth iteration when object pooling was added, contributing to the decrease in

maintainability. Optimally, the pooling behavior should be refactored to its own class due to the importance of managing complexity and ensuring proper design and abstraction when adding new features to avoid negatively impacting maintainability.

Other classes as MeleeEnemy, BulletBehavior, and PlayerCamera exhibit relatively consistent maintainability scores across all iterations due to the flexibility of interfaces in the classes. Especially the BulletBehavior maintained a good design since it was reused for multiple weapons without complications.

DOD

In regards to the DOD implementation, the results showed that the overall maintainability was better than the OOD implementation for the first iteration due to its simple design and responsibility distribution. Each system had a well-defined responsibility with each system being executed independently of each other. But gradually decrease when more complex features were added. The overall maintainability score ended up being slightly lower for this implementation compared to the OOD implementation. This is mainly caused by two factors: the codebase containing more lines of code and the cyclomatic complexity being higher due to lack of responsibility distribution. Especially the EnemyBehaviorSystem and EnemySpawnerSystem ended up being responsible for multiple features which breaks the principle of single responsibility.

The EnemySpawnerSystem, EnemyBehaviorSystem, and BulletSpawnerSystem start with high maintainability scores but gradually decrease in subsequent iterations. The decrease can be attributed to the increased complexity and responsibilities added to these systems over time. Especially the introduction of object pooling in the fifth iteration increased the complexity of the EnemySpawnerSystem and EnemyBehaviorSystem. The addition of complex logic to these systems significantly reduced the maintainability score. To improve maintainability, the design of these systems should be reviewed to encourage modularized responsibilities. By breaking down the complex logic into smaller components, the systems can be made less complex and more maintainable.

Other systems like the DayNightSystem, LootBehaviorSystem, and LootSpawnerSystem maintain a moderate level of maintainability throughout all iterations.

A notable mention in regard to calculating the maintainability index is the size of the codebase. Section 8.4 analyzed the size difference between the two codebases which showed that the DOD codebase was significantly larger in comparison to the OOD implementation. This affects the results from the maintainability index calculations that are based on lines of code alongside other parameters. Therefore, it is not possible to fully evaluate the maintainability value based on these scores, it should be evaluated alongside the cohesion and coupling analysis in order to make a reasonable argument.

9.1.4 Code Change and Development Time

The analysis in Section 8.4 on code change and development time revealed that the DOD version consistently had a larger codebase compared to the OOD version across all iterations. The DOD version is approximately three times larger than the OOD version throughout the project which is caused by the necessary overhead of defining components, attaching these systems to specific game objects that are later converted to entities, and defining jobs to be executed in the update loops in the systems. The larger codebase affects the complexity of the project making it slightly more difficult for developers to understand since they have to maintain knowledge of more lines of code. Therefore, a clear and understandable code organization is crucial to keep the maintainability level high. To stay organized developers must follow the single responsibility principle and have consistent naming conventions due to the flat structure that DOD promotes.

The development time showed some interesting results. It showed that the DOD implementation required significantly more development time during the initial implementation, due to the additional setup required to make the application work as intended. Especially the player and camera movement required much attention due to the complexity of the camera not being able entity convertible. Also, the second and third iterations required more development time due to the addition of complex features. However, the fourth and fifth iterations were completed faster than the OOD version. These iterations for the DOD version were mainly code modifications which proved to take less time. Especially the fifth iteration only required minor changes to the existing behavior in the DOD version, while it required more work in the OOD version. From this, it is possible to argue that the flexibility of the DOD implementation was better when considering modifications rather than additions.

However, it is worth mentioning that the experimental setup required the development of both versions within the same week. At each iteration, this was executed by implementing the OOD version first and hereafter the DOD version. Therefore, it is possible to argue that the development time has been affected by the knowledge gained from implementing the OOD version before the DOD version. Optimally, the order of development should have been changed by switching randomly between implementing OOD and DOD first. Another improvement to the experimental setup would be to have two different teams develop the two versions. This way, the study would not be biased during the implementation phase. Unfortunately, this was not possible due to the limited resources of this project.

9.1.5 Summary

The comparison between the OOD and DOD game versions revealed several key findings. Both designs had some cohesion issues in certain components, such

as the Enemy class in the OOD version and the EnemyBehaviorSystem in the DOD version. A tendency to add too many features to some systems in the DOD version was observed. This made them more complex and difficult to understand in comparison to the OOD version. DOD required more development time in the initial iterations and when adding features. Both designs exhibited increased dependencies and coupling in the third iteration. For the DOD version, this was mainly caused due to lack of responsibility distribution, and could easily be eliminated with minor code refactoring. However, the nature of DOD emphasized greater decoupling of the code, making it easier and faster to make modifications to existing systems.

The findings indicate that it is easier and faster to make quick iterations and experiments with the OOD version but as the codebase grows larger the DOD version exhibits greater flexibility due to its data-oriented nature. The DOD version allowed for easier modification of features, resulting in a more adaptable and flexible implementation. In contrast, the OOD version had more inflexible structures, resulting in tighter coupling and more dependencies between classes which limited its flexibility.

9.2 Scalability

From the results presented in Section 8 information on frame rate and CPU usage time was collected in order to validate the scalability of the two game versions.

While FPS and CPU usage time are closely related it was still relevant to measure both metrics due to the external factors that can affect FPS. The results showed a clear difference between the two versions.

In OOD, the emphasis is on organizing the game functionality into objects, which encapsulate both data and behavior. While OOD provides concepts like encapsulation, inheritance, and polymorphism to structure the codebase and make it more flexible, it does not address scalability concerns. The results showed that FPS drops quickly when more enemies are added, while the CPU time increases. This is caused by the memory layout and the use of encapsulation that led to indirect memory access patterns as explained in Section 2. It is worth mentioning that the FPS and CPU usage time did not change drastically when more complex behavior was added. For example, if we look at Figure 20 we can see that the FPS is within an acceptable range for the case where 100 enemies were spawned. This indicates that scalability is not related to the additional logic implemented but rather the number of enemies needed to be iterated through when performing transformations on the data.

Another interesting finding was observed during the analysis. The introduction of object pooling in iteration five slightly improved both the FPS and CPU usage time for the OOD version. However, it significantly decreased the per-

formance of the DOD version. Through profiling and code inspection, it was discovered that inefficient memory allocation was the problem for the DOD version. In the EnemyBehaviorSystem a list of all available throwables will be constructed, used, and deconstructed each frame. This significantly impacts the performance of the game. While the OOD version contains a global list, which is allocated once and update accordingly resulting in a slight improvement.

Overall, the results clearly showed that the DOD version is more scalable in comparison to the OOD version. The gathered performance metrics indicate that the performance decreases exponentially for the OOD version and linearly for the DOD version when more enemies are added to the game.

Furthermore, there is evidence that the use of hardware has a significant impact on scalability. For both versions, we observed that PC2 performed almost twice as well compared to PC1 in terms of both FPS and CPU usage time. This is correlated to the fact that PC2 had better specifications compared to PC1. PC2 had a faster core speed, more cores, and more threads, all contributing to the increased performance. This proves the importance of hardware knowledge when developing games. Imagine a game created for PlayStation 4. In this case, the developers have complete knowledge of all hardware specifications enabling a complete performance overview for the developers. This is difficult to obtain when creating games for PC where hardware specification varies much more. Developers must optimize their code in order to create a scalable game being playable on a wide range of hardware. The results showed that DOD can aid this process and make games more scalable.

9.3 Design Principles

While adhering to the SRP is a common approach to achieving code flexibility, it is essential to acknowledge the importance of other principles as well. The SOLID principles[37], which encompasses a set of guidelines, offer further practices for flexibility when properly applied. These principles are commonly referenced in OOD. The initial principle, as previously discussed, is SRP.

The second principle is the Open-Closed Principle (OCP), which emphasizes the need for an implementation to accommodate new features or modifications without necessitating changes to existing code. In the OOD implementation, the code has been designed to uphold this principle to some extent. An example of this can be observed in the Weapon class, where the addition of new weapon types is facilitated without requiring modifications to other classes. However, the Enemy class encountered difficulties in adhering to this principle, as it had to undergo modifications in the second iteration to incorporate the RangeEnemy class.

The third principle is the Liskov Substitution Principle (LSP), which states that derived classes should be capable of substituting their base classes while adhering to the defined behavior of the base class. This principle has been successfully implemented in the OOD version by introducing base classes for enemy and weapon types, thereby establishing common behavior and eliminating the

need for redundant implementations.

The fourth principle is the Interface Segregation Principle (ISP), which states that classes should not be forced to depend on interfaces that they do not utilize. Although this principle has been partially adhered to, it lacks attention in the Enemy class, for instance. It is evident that the class assumes multiple responsibilities while functioning as a base class. Consequently, enemy types that do not require the attacking behavior are still reliant on the interface that defines this behavior. To address this, the behavior should be divided into multiple smaller interfaces.

The final principle is the Dependency Inversion Principle (DIP), which advocates for decoupling through abstraction rather than direct instantiation within classes. The implementation lacks attention to this principle, as it solely relies on direct access throughout the code.

A noteworthy aspect regarding principles such as SOLID is the limited attention given to common software practices in the current literature on DOD. While principles like SRP and OCP can be applied, the relevance of LCP, ISP, and DIP is lessened, as derived classes, interfaces, and dependencies are not commonly employed within the DOD context. Such abstractions are not inherent to the general design paradigm. Consequently, developers are confronted with basic principles such as the separation of data and behavior, as well as SRP, but they lack comprehensive guidance on system structure and design best practices. This lack of guidance can lead developers, particularly those lacking experience or under time constraints, to create large systems that encompass numerous behaviors, potentially resulting in rushed and incautious decision-making.

10 Future Work

This thesis conducted a case study on flexibility and scalability in game development. While the results of this thesis were successful showing that DOD provides both great flexibility and scalability in comparison to OOD, future work could improve the results and validity of the study.

Firstly, the two game versions were implemented by the writer of this paper within a relatively short time frame. To reduce potential biases and increase the validity of the results, it would be beneficial to involve two independent development teams in future work. Having two teams responsible for implementing the OOD and DOD version of the game, the study would be able to capture a broader range of development perspectives. This approach would better reflect real-world scenarios where game development often involves collaboration among multiple developers.

Furthermore, extended development time beyond one week per iteration would contribute to a more thorough and comprehensive implementation. The rushed development may have led to sub-optimal design decisions, such as the missing observer pattern for the OOD implementation. Additional time for each iteration would enable a more mindful development process.

Secondly, the code analysis for the flexibility part could be improved. The case study mainly employed manual code inspection alongside metrics calculation tools. Incorporating an expert group to evaluate the quality of the code would further enhance the validity of the analysis. By having an expert group inspect the code for each iteration, factors such as code readability, maintainability, and architectural design could be more accurately evaluated. We would be able to provide a more nuanced perspective on the strengths and weaknesses of our implementations in terms of code quality.

Another measure of flexibility, which has not been explored in this thesis, is testability. Testability refers to the ease with which code can be tested[13]. This encompasses various factors, such as the ability to isolate parts of the codebase for testing, establishing test environments, and verifying the correctness of logical operations. Testability and code flexibility are closely related since a flexible implementation should emphasize modularity and independence, thereby allowing for independent and isolated testing of different components. A codebase that is highly testable suggests that it has a well-structured and modular design, with clear separations between components. Therefore, this enhanced modularity simplifies code modifications, as changes made to one component are less likely to impact others.

11 Conclusion

The first objective of this thesis was to contribute with a historical overview of the events that led to the popularity of data-oriented design (DOD) in game development. This thesis provided a comprehensive historical overview of the evolution in game development methodologies from the early days of video games when developers had to be highly hardware-oriented due to limited hardware specifications, to the increasing popularity of object-oriented design (OOD) that provided greater flexibility, and lastly how OOD impacted performance forcing developers to turn back to data-oriented principles to keep up with player demands.

The second objective of this thesis was to address the gap in the literature regarding the impact of DOD on flexibility and scalability in game development. This was achieved through a comparative case study in which two versions of the same game were implemented with OOD and DOD. By analyzing the effects of the two design paradigms through multiple implementation iterations, valuable insights into the importance of flexibility and scalability in game development have been provided.

The conducted research revealed that DOD offers significant advantages in terms of flexibility and scalability compared to OOD. The improved cache efficiency, reduced memory overhead and increased parallelism provided by DOD contribute to more scalable game development. These factors enable developers

to create games that are able to handle larger amounts of data that leverage modern hardware architectures.

While both versions had their strength and weaknesses in terms of flexibility, the results showed that DOD provided greater decoupling between systems, allowing developers to utilize a component-based architecture that allowed for easier adaptation of change requirements and decreased development time.

Furthermore, the study highlighted the importance of data and hardware awareness in game development in which developers must make careful considerations in regards to the tradeoffs between flexibility and scalability when creating games.

In conclusion, data and hardware-based challenges are only likely to escalate due to the increased player demands. Therefore, the necessity to adopt data-oriented design strategies will increase to allow developers to make flexible and scalable games in the future.

Bibliography

[1] Ishfaq Ahmad, Sanjay Ranka, and Samee Ullah Khan. "Using game theory for scheduling tasks on multi-core processors for simultaneous optimization of performance and energy". In: *2008 IEEE international symposium on parallel and distributed processing*. IEEE. 2008, pp. 1–6.

[2] Gerald Aigner and Urs Hölzle. "Eliminating virtual function calls in C++ programs". In: *ECOOP'96—Object-Oriented Programming: 10th European Conference Linz, Austria, July 8–12, 1996 Proceedings 10*. Springer. 1996, pp. 142–166.

[3] K Fedoseev et al. "Application of Data-Oriented Design in Game Development". In: *Journal of Physics: Conference Series* 1694.1 (2020). DOI: 10.1088/1742-6596/1694/1/012035.

[5] Apostolos Ampatzoglou and Alexander Chatzigeorgiou. "Evaluation of object-oriented design patterns in game development". In: *Information and Software Technology* 49.5 (2007), pp. 445–454. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2006.07.003. URL: https://www.sciencedirect.com/science/article/pii/S0950584906000929.

[7] Imran Baig. "Measuring Cohesion and Coupling of Object-Oriented Systems - Derivation and Mutual Study of Cohesion and Coupling". In: (2004). Accessed on May 1, 2023.

[8] Jessica D Bayliss. "The data-oriented design process for game development". In: *Computer* 55.05 (2022), pp. 31–38.

[9] William R Bitman. "Balancing software composition and inheritance to improve reusability, cost, and error rate". In: *Johns Hopkins APL Technical Digest* 18.4 (1997), pp. 485–500.

[10] Carlos Carvalho. "The gap between processor and memory speeds". In: *Proc. of IEEE International Conference on Control and Automation*. Vol. 5000. 10000. 2002, p. 15000.

[11] Dawid Cieżarkiewicz. "The Faster You Unlearn OOP, the Better for You and Your Software". In: *GameDev.net* (Dec. 2018). URL: https://www.gamedev.net/articles/programming/general-and-gameplay-programming/the-faster-you-unlearn-oop-the-better-for-you-and-your-software-r5026/.

[12] Lucas Coulson. "Game Development on Multicore". In: (Aug. 2022). URL: https://nationalpcbuilder.com/the-difference-between-30-fps-and-60-fps-when-gaming/.

[13] D Davis. "Modelled on software engineering: flexible parametric models in the practice of architecture (2013)". In: *RMIT University* ().

[14] Harpal Dhama. "Quantitative models of cohesion and coupling in software". In: *Journal of Systems and Software* 29.1 (1995), pp. 65–74.

[15] Dino Dini. *Beam Me Up, Scotty!* https://dinodini.wordpress.com/2010/12/03/beam-me-up-scotty/. Accessed on May 10. 2023. Dec. 2010.

[16] Wolfgang Engel. *GPU Pro 4: Advanced Rendering Techniques*. Boca Raton, FL: CRC Press, 2013. ISBN: 9781466567436.

[17] Björn Eriksson and Maria Tatarian. *Evaluation of CPU and Memory performance between Object-oriented Design and Data-oriented Design in Mobile games*. 2021.

[18] Andrés Alberto Estevez and Augusto Salgado. "A Component-Based Architecture for Unity". In: *Etermax Technology Blog* (Jan. 2023). Accessed on May 1, 2023. URL: https://medium.com/etermax-technology/a-component-based-architecture-for-unity-ff211ca478fe.

[19] Richard Fabian. "Data-oriented design". In: *framework* 21 (2018), pp. 1–7.

[20] Daniel Graziotin. "Object Oriented Memory Management (Java and C++)". In: (). Accessed 18. May, 2023. URL: https://ineed.coffee/uploads/object-oriented-memory-management-java-c++.pdf.

[21] Toni Härkönen. "Advantages and Implementation of Entity-Component-Systems". In: (2019).

[22] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[23] Kate Holderer. "Single Responsibility Principle: Software Design". In: *POGIL Activity Clearinghouse* 3.4 (2022).

[24] Abhijeet Kale. *CPU Utilization in Performance of Computer Systems*. https://www.linkedin.com/pulse/cpu-utilization-performance-computer-systems-abhijeet-kale/. Accessed: May 6, 2023. 2020.

[25] Ken Kennedy and Kathryn S McKinley. "Optimizing for parallelism and data locality". In: *Proceedings of the 6th international conference on Supercomputing*. 1992, pp. 323–334.

[27] Noel Llopis. "Data-oriented design (or why you might be shooting yourself in the foot with OOP)". In: *Game Developer Magazine* 16.8 (2009).

[28] Microsoft. *Code Metrics Values in Visual Studio*. https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2022. Accessed: May 1, 2023. 2022.

[31] Robert Nystrom. *Game programming patterns*. Genever Benning, 2014.

[32] John K Ousterhout. *A philosophy of software design*. Vol. 98. Yaknyam Press Palo Alto, CA, USA, 2018.

[33] Tekla S Perry and Paul Wallich. "Microprocessors: Design case history: The atari video computer system: By omitting lots of hardware, designers added flexibility and gave video-game programmers room to be creative". In: *IEEE Spectrum* 20.3 (1983), pp. 45–51.

[35] Matyáš Racek. "Data Oriented Design is not ECS". In: (2020). Accessed: May 9, 2023. URL: https://www.gamedeveloper.com/design/the-entity-component-system---an-awesome-game-design-pattern-in-c-part-1-.

[36] Markus Rapp. "The Difference Between 30 FPS and 60 FPS When Gaming". In: (Jan. 2012). URL: http://www.markusrapp.de/projects/game-development-on-multi-core/.

[37] Harmeet Singh and Syed Imtiyaz Hassan. "Effect of solid design principles on quality of software: An empirical assessment". In: *International Journal of Scientific & Engineering Research* 6.4 (2015), pp. 1321–1324.

[38] Tobias Stein. "The Entity-Component-System - An awesome game-design pattern in C++ (Part 1)". In: (2017).

[39] Steve. *Atari 2600 Hardware Design: Making Something out of (Almost) Nothing*. https://www.bigmessowires.com/2023/01/11/atari-2600-hardware-design-making-something-out-of-almost-nothing/. Accessed on 20. May 2023. Jan. 2023.

[40] TechBullion. "The History of C++ Coding in Video Games". In: *TechBullion* (Mar. 2023). URL: https://techbullion.com/the-history-of-c-coding-in-video-games/.

[41] Techopedia. *Frames Per Second (FPS)*. https://www.techopedia.com/definition/7297/frames-per-second-fps. Accessed on May 1, 2023. 2022.

[42] Unity. "Unity Manual: Entities package". In: (2023). Accessed: May 9, 2023. URL: https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/index.html.

[43] Anne Van Ede. "ENCODE: From Object-Oriented to Data-Oriented, an Automatic ECS Conversion Designer and Education Tool". In: (July 2021). URL: https://github.com/AnneVanEde/MasterThesis/tree/main.

[44] Andrew Williams. *History of digital games: developments in art, design and interaction*. CRC Press, 2017.

[45] Wayne Wolf. *High-performance embedded computing: architectures, applications, and methodologies*. Elsevier, 2010.

Ludography

[4] Alda Games. *Zombero: Archero Hero Shooter*. Version 1.14.4. Aug. 24, 2020. URL: https://play.google.com/store/apps/details?id=com.aldagames.zombero.bullet.hell&hl=en_US.

[6] Atari. *Pong*. Version 1.0.0. Nov. 29, 1972. URL: https://ponggame.io/.

[26] Kongregate Inc. *Survivor.io*. Version 1.13.1. Oct. 11, 2017. URL: https://play.google.com/store/apps/details?id=com.dxx.firenow&hl=en_US.

[29] Nintendo. *Excitebike*. Version 1.0.0. Nov. 28, 1984. URL: https://www.retrogames.cz/play_055-NES.php.

[30] Nintendo. *Super Mario Bros*. Version 1.0.0. Sept. 13, 1985. URL: https://supermarioplay.com/.

[34] Poncle. *Vampire Survivors*. Version 0.1.0. Mar. 31, 2021. URL: https://store.steampowered.com/app/1794680/Vampire_Survivors/.