

Universidad San Jorge

Escuela de Arquitectura y Tecnología

**Grado en Diseño y Desarrollo de
Videojuegos**

Proyecto Final

**Comparing Object Oriented and Data Oriented
Programming for video games**

Autor del proyecto: Daniel Muñoz Muñoz

Director del proyecto: Daniel Blasco

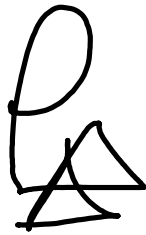
Zaragoza, 26 de junio de 2023



Este trabajo constituye parte de mi candidatura para la obtención del título de Graduado en Ingeniería Informática por la Universidad San Jorge y no ha sido entregado previamente (o simultáneamente) para la obtención de cualquier otro título.

Este documento es el resultado de mi propio trabajo, excepto donde de otra manera esté indicado y referido.

Doy mi consentimiento para que se archive este trabajo en la biblioteca universitaria de Universidad San Jorge, donde se puede facilitar su consulta.

A handwritten signature in black ink, consisting of a large loop at the top and several intersecting lines below it.

Firma

Fecha

26 de junio de 2023

Dedicatoria y Agradecimiento

Me gustaría agradecer a mi familia por apoyarme durante todos estos años de carrera, viéndome en las mañanas y en las buenas, por brindarme la oportunidad de estudiar lo que me apasiona desde pequeño y ayudarme a descubrir mi pasión. Gracias a vosotros siento que este trabajo tiene sentido más allá de mi formación.

También quiero agradecer a mis amigos/as por haber estado ahí cuando no sé pedía y por haberme motivado a seguir adelante con apoyo y ánimos. En especial quiero agradecer a Mercedes Gil (Merchi), su compañía en mi vida me ha dado fuerzas en los días que sentía que no podía levantarme de la cama y siempre ha estado apoyándome sin pedir nada a cambio. Siento que el trabajo, la dedicación y el amor que tengo por lo que hago no sería lo mismo sin su presencia en mi vida.

Quiero particularmente darle las gracias a Daniel Blasco por confiar y apoyarme a lo largo de todo el proyecto con su interés y conocimiento para que no solo este proyecto alcanzara su mejor versión, sino también para que yo lo hiciera y aprendiera lo máximo posible.

Junto a él tengo que mencionar a África Domingo, mi tutora desde que entre a la universidad. Su apoyo constante, su interés y ayuda me ha hecho sentir en casa desde que entre en la universidad, haciendo sentirme de una forma que nunca había hecho.

Al resto de la Universidad San Jorge y en especial al equipo de investigación SVIT por brindarme una oportunidad como nunca antes haciéndome sentir en casa y empujándome para dar lo mejor de mí cada día que pasa

Gracias a todos por haberme ayudado a ser quien soy ahora.

Table of contents

Abstract	1
Resumen	2
1. Introducción	3
1.1. Object Oriented Design for video games	3
1.2. Data-Oriented Designs for video games	4
1.3. Motivation	5
1.4. Industrial application: Introducing Bullet Hells	7
2. State of Art	9
2.1. Overview of Object-Oriented Design (OOD)	9
2.1.1. <i>Definition and principles of OOD</i>	10
2.1.2. <i>Application of OOD in game development</i>	11
2.1.3. <i>Advantages and limitations of OOD in game projects</i>	12
2.2. Introduction to Data-Oriented Design (DOD)	13
2.2.1. <i>Definition and core principles of DOD</i>	14
2.2.2. <i>Contrasting DOD with OOD in design philosophy and approach</i>	15
2.2.3. <i>Case studies showcasing implementations of DOD in game development</i>	16
2.3. Comparative Analysis of OOD and DOD	17
2.3.1. <i>Key differences between OOD and DOD from a learning point of view</i>	17
2.3.2. <i>Evaluation of Strengths and Weaknesses in the game development context</i>	18
2.4. Industry Practices and Trends	18
2.4.1. <i>Use of OOD and DOD in the game development industry</i>	19
2.4.2. <i>Case study: Unity's efforts to develop DOTS</i>	20
2.4.3. <i>Comparing the learning curve of OOD and DOD</i>	22
2.4.4. <i>Discussion of emerging trends and hybrid approaches combining OOD and DOD</i>	22
2.5. The current state of the Bullet Hell genre	23
3. Objectives	25
4. Methodology	26
4.1. Development methodology	26
4.2. Evaluation methodology	28
4.2.1. <i>Research Questions</i>	28

4.2.2.	<i>Metrics</i>	29
4.2.3.	<i>Data Collection</i>	29
4.2.4.	<i>Data Analysis</i>	29
4.3.	Methodology choice	29
4.4.	Working routines	31
5.	Description and Implementation	32
5.1.	Game description and mechanics	32
5.2.	OOD implementation	33
5.2.1.	<i>Implementing the code</i>	34
5.2.2.	<i>Interactions and pipeline</i>	35
5.2.3.	<i>Script division</i>	36
5.2.4.	<i>In-Editor view</i>	41
5.3.	DOD implementation	43
5.3.1.	<i>Understanding the framework</i>	43
5.3.2.	<i>Creating the data worksheet</i>	45
5.3.3.	<i>Implementing the code</i>	47
5.3.4.	<i>Pipeline</i>	47
5.3.5.	<i>Script division</i>	49
5.3.6.	<i>In-Editor view</i>	55
6.	Economic Study	59
6.1.	Cost analysis	59
7.	Results	61
7.1.	Prototype review	61
7.2.	Comparative analysis	64
7.2.1.	<i>Bullet stress test</i>	64
7.2.2.	<i>Enemy stress test</i>	67
8.	Validation Threats	69
9.	Conclusions	71
9.1.	Future Work	72
9.2.	Personal Learnings	74
10.	References	75
11.	Appendix	82

11.1. Appendix A	82
<i>11.1.1. Datasets.....</i>	<i>82</i>
11.2. Appendix B	83
<i>11.2.1. Project files and builds</i>	<i>83</i>
12. Annexes:.....	84
12.1. Annex A: Propuesta de proyecto.....	84
12.2. Annex B: Meetings record	85
12.3. Annex C	86
<i>12.3.1. Data worksheet</i>	<i>86</i>
12.4. Annex D	87
<i>12.4.1. Time description per script</i>	<i>87</i>
12.5. Annex E	88
<i>12.5.1. Gallery</i>	<i>88</i>
12.6. Annex F.....	90
<i>12.6.1. Data Collection: Unity's profiler.....</i>	<i>90</i>

Table of Figures

Figure 1: Breakdown of the video game market by platform [3].....	3
Figure 2: Representation of a complex world [11].....	4
Figure 3: Abstract representation [11].....	4
Figure 4: DOTS Framework and Packages [29].....	5
Figure 5: Summary of DOD usage over time.....	7
Figure 6: DoDonPachi Screenshot [37].....	8
Figure 7: Touhou Project: Ancient New Moon [36].....	8
Figure 8: OOP representation of data distribution [60].....	13
Figure 9: Performance comparison depending on cache misses [14].....	15
Figure 10: Performance comparison depending on data status [14].....	15
Figure 11: DOD representation of data distribution [60].....	16
Figure 12: ECS infographic [23].....	20
Figure 13: Burst view of assembly code [89].....	21
Figure 14: Fantastic Poetry Festival, created with Danmoku [96].....	24
Figure 15: Scheme application of the GQM methodology.....	30
Figure 16: Structure for the Object Oriented Structure.....	34
Figure 17: Bullet relationships.....	36
Figure 18: Player relationships.....	38
Figure 19: Enemy relationships.....	39
Figure 20: Definition & Generic relationships.....	40
Figure 21: OOD Hierarchy view.....	41
Figure 22: Enemy manager parameters.....	41
Figure 23: Player manager parameters.....	41
Figure 24: Bullet parameters.....	42
Figure 25: General definition of ECS [101].....	43
Figure 26: Internal definition of an entity [101].....	43
Figure 27: Entity grouping [101].....	44
Figure 28: Archetype definition inside a world [101].....	44
Figure 29: Structure for the Data Oriented Structure.....	47
Figure 30: Job branching dependencies [103].....	48
Figure 31: Bullets processing relationship.....	49
Figure 32: Bullet authoring script.....	50
Figure 33: Player processing relationship.....	51
Figure 34: Enemy processing relationship.....	52

Figure 35: Definition relationship	53
Figure 36: Authoring components	55
Figure 37: Basic Unity hierarchy	55
Figure 38: Configuration authoring	55
Figure 39: Entities Hierarchy window	56
Figure 40: Components window and Pierce inspector	56
Figure 41: System window	58
Figure 42: Archetype window and Bullet inspector	57
Figure 43: OOD Prototype look	61
Figure 44: Showcasing of some bullet patterns heading for the enemies	62
Figure 45: Comparison of players. OOD on the left, DOD on the right	62
Figure 46: OOD runtime performance	63
Figure 47: DOD runtime performance	63
Figure 48: Frames per second as the number of bullets increases.....	64
Figure 49: ms as the number of bullets increases for managed scripts	64
Figure 50: ms as the number of bullets increases for the whole frame	65
Figure 51: ms as the number of bullets increases for the managed scripts with new version ..	65
Figure 52: Final results for bullets and in-game times	66
Figure 53: MB of data allocated as the number of bullets increases	66
Figure 54: ms change over managed scripts as enemies increase	67
Figure 55: ms change over the full frame as enemies increase	67
Figure 56: MB of data allocated as the number of enemies increases.....	68

Table of Tables

Table 1: Data worksheet for bullet entities	45
Table 2: Data worksheet for player entities	45
Table 3: Data worksheet for enemy entities	45
Table 4: Data worksheet component transformations	46
Table 5: Costs of the project by hours	59

Abstract

Object-Oriented Programming (OOP) and Data-Oriented Programming (DOP) are two prominent design paradigms widely used in software development, including video game development. This research aims to compare these paradigms specifically in the context of the video game environment.

OOP focuses on modeling systems as a collection of interacting objects, encapsulating data and behavior within classes. It promotes modularity, code reusability, and maintainability, making it suitable for various aspects of game development. In video games, OOP facilitates the creation of game entities such as characters, items, and environments and supports features like inheritance and polymorphism to manage complexity and enable efficient game object interactions.

DOP, on the other hand, emphasizes organizing and optimizing data for efficient processing. It seeks to maximize data locality, minimize cache misses, and exploit parallelism for performance gains. DOP techniques are particularly beneficial in in-game scenarios where data-oriented optimizations such as physics simulations, AI processing, and rendering can significantly improve performance.

Through this comparative analysis, the research aims to provide insights into the trade-offs and suitability of OOP and DOP in different aspects of video game development. By understanding their relative merits, developers can make informed decisions regarding the choice of design paradigm based on the specific requirements and constraints of their game projects.

This study focuses on a case study representing a characteristic idiosyncrasy of a video game to conduct comparisons by developing a prototype video game using both OOP and DOP approaches.

The results show that projects, where performance is essential, can benefit significantly from DOD, while it does not impact small or medium-sized projects as much. The current literature and educational offer are more oriented toward OOD training than DOD. Therefore, the mass adoption of DOD by the community will depend on whether said balance changes due to its successful use in industrial products and research works, along with which this research tries to contribute.

Keywords

Design paradigms, data-oriented design, object-oriented design, performance optimization, game engine architecture, game performance, data locality, cache optimization, parallel processing, code optimization, code maintainability, and code reusability.

Resumen

La Programación Orientada a Objetos (OOP) y la Programación Orientada a Datos (DOP) son dos paradigmas de diseño prominentes ampliamente utilizados en el desarrollo de software, incluido el desarrollo de videojuegos.

OOP se enfoca en modelar sistemas como una colección de objetos que interactúan, encapsulando datos y comportamiento dentro de clases. Promueve la modularidad, la reutilización del código y la mantenibilidad. En el desarrollo de videojuegos, OOP facilita la creación de entidades de juegos, como personajes, elementos y entornos, y admite funciones como la herencia y el polimorfismo para administrar la complejidad y permitir interacciones eficientes con los objetos del juego.

DOP, por otro lado, enfatiza la organización y optimización de datos para un procesamiento eficiente. Busca maximizar la localidad de los datos, minimizar las fallas de caché y explotar el paralelismo para mejorar el rendimiento. En los videojuegos, las técnicas DOP son beneficiosas para escenarios donde las optimizaciones orientadas a datos pueden mejorar significativamente el rendimiento, como simulaciones físicas, procesamiento de IA y renderizado.

A través de este análisis comparativo, el proyecto tiene como objetivo proporcionar información sobre los beneficios de OOP y DOP en diferentes aspectos del desarrollo de videojuegos. Al comprender sus diversos usos y beneficios, los desarrolladores pueden tomar decisiones informadas con respecto a que paradigma de diseño usar en función de los requisitos y limitaciones específicos de sus videojuegos.

Este estudio se centra en un caso de estudio que representa la idiosincrasia característica de un videojuego para llevar a cabo las comparaciones mediante el desarrollo de un prototipo de videojuego utilizando ambos paradigmas.

Los resultados muestran que los proyectos en los que el rendimiento es esencial pueden beneficiarse notablemente de DOD, mientras que no tiene tanto impacto en proyectos de tamaño medio o pequeño. La literatura y oferta educativa actual están más orientadas a la formación en OOD frente a DOD, por ello, la adopción masiva de DOD por la comunidad dependerá de que dicho equilibrio cambie debido a su uso exitoso en productos industriales y trabajos de investigación, a lo cual el presente trabajo trata de contribuir.

Palabras Clave

Paradigmas de diseño, diseño orientado a datos, diseño orientado a objetos, mejora del rendimiento, arquitectura del motor de juego, rendimiento del juego, estructuración de datos, optimización de caché, paralelización, optimización de código, mantenibilidad de código, reusabilidad de código.

1. Introducción

The video game industry has witnessed remarkable growth and innovation in recent years, with games becoming increasingly complex and immersive [1] [2] [3]. This rapid evolution has brought new challenges for game designers and developers, particularly in performance optimization, memory management, and scalability, requiring research to find the optimal patterns for the task [4] [5].

This rapid evolution has only gotten more attention recently with the uprise in mobile games and their user popularity in the overall gaming market (**Figure 1**). These demands have pushed for better hardware, but it does come with an upgrading cost that not all users can afford, which demands a development apart from just hardware [6] [7] [8].

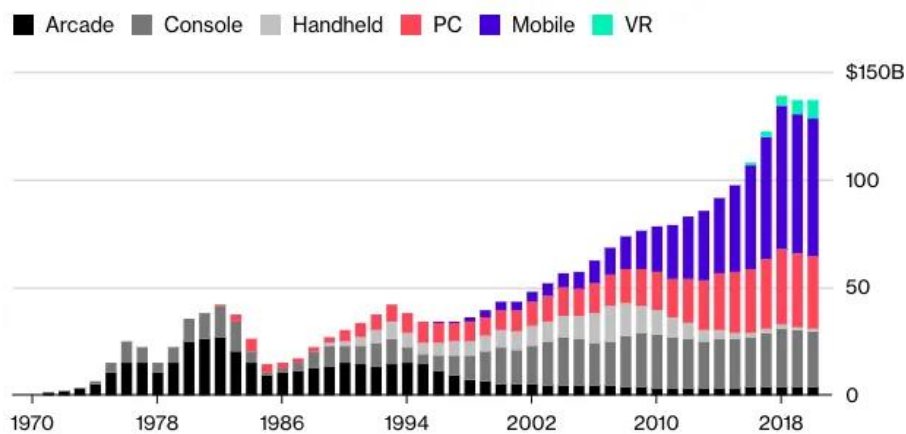


Figure 1: Breakdown of the video game market by platform [3]

This has pushed some game engines to search for other solutions to improve performance in the general picture, from indie developers to AAA developers. While sharing the same problem, the two most significant engines [9] have had different approaches to easing them. Unreal Engine has opted for a new system called "Nanite," which eases the work on the GPU and CPU [10]. On the other hand, Unity has not officially launched anything yet but has been experimenting since 2018 with a complete change of coding paradigm from its engine to the programmers using it.

1.1. Object Oriented Design for video games

Object-oriented programmers can recreate complex worlds within video games through the power of abstraction (**Figure 2**). By representing real-world entities and concepts as classes (objects) in their code, they can encapsulate the characteristics, behaviors, and relationships of various elements that make up a game environment without getting into the intrinsics of how

things work or are managed (**Figure 3**). For example, they can define characters, objects, environments, and interactions classes, allowing them to model the virtual world with high fidelity. This abstraction enables programmers to organize and manage the game's components efficiently, as each class encapsulates its logic and data, promoting modularity and code reusability.

Unity's change is promoted by most games being designed with Object Orientation in mind described by Nystrom R. [12]. A common reason for this is that its use is easy to maintain and find by the developers, thus making it cheaper and faster to develop.

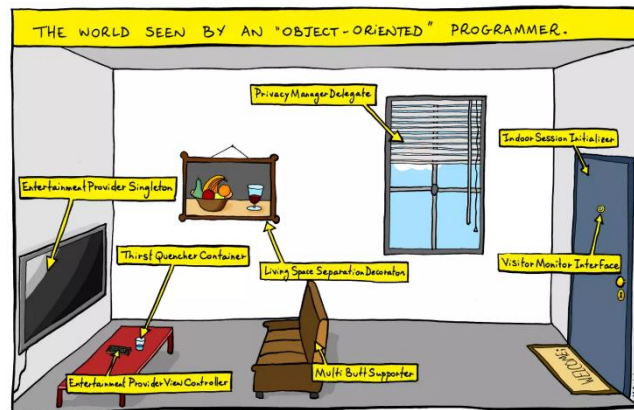


Figure 2: Representation of a complex world [11]

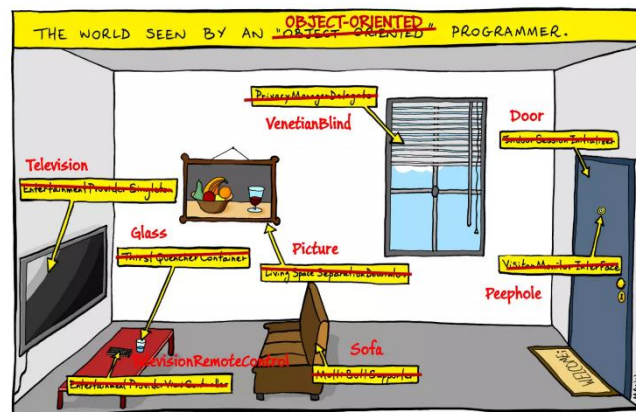


Figure 3: Abstract representation [11]

Object Oriented Design (OOD) standard design patterns apply to adaptive games with templates like Command, Observer, Flyweight, and Singletons, which have been used to help with many video game tasks [13]. While these patterns are both valuable and maintainable, there are more optimal options for performance. An example is how Singletons help with readability and usability along a bigger scheme. However, it does not use memory accordingly as it usually must jump through the caches (short time memory collections of growing size) to reach it, leading to lower performances.

1.2. Data-Oriented Designs for video games

The origin of Data-Oriented Design (DOD) comes from targeting data for operations [14] [15] [16] with a focus on data locality (accessing data as fast as possible) and parallel computing [14] [17] [18] to improve the performance of those operations.

The main application of DOD has been in real-time systems, as they require stable performance under limited resources [19]. One of the key uses has been on video recordings as it must quickly process data and store it without losing frames, as recovering it would delay the process and the

recording. While hardware improvements have helped, DOD has proven to have a consistent performance gain over OOP as they have a lower chance of page skipping (event at which a given instruction or variable is not on the currently available memory section, so we need to go to main memory to retrieve it), which would lead to freezes between memory and disk [20] [21].

DOD has not been used in the video games industry as it is believed to require much effort from developers [22]; Unity has pushed for it since 2018 with their new framework called DOTS (Figure 4), which uses ECS, a programming paradigm based on DOD [23]. This system has been used for a few prototype games [24] [25] [26] [27] [28] [29], and some bigger ones are near production ready [30]. However, the technology is mainly targeted to processor-intensive games with lots of rendering, physics, and many calculations.

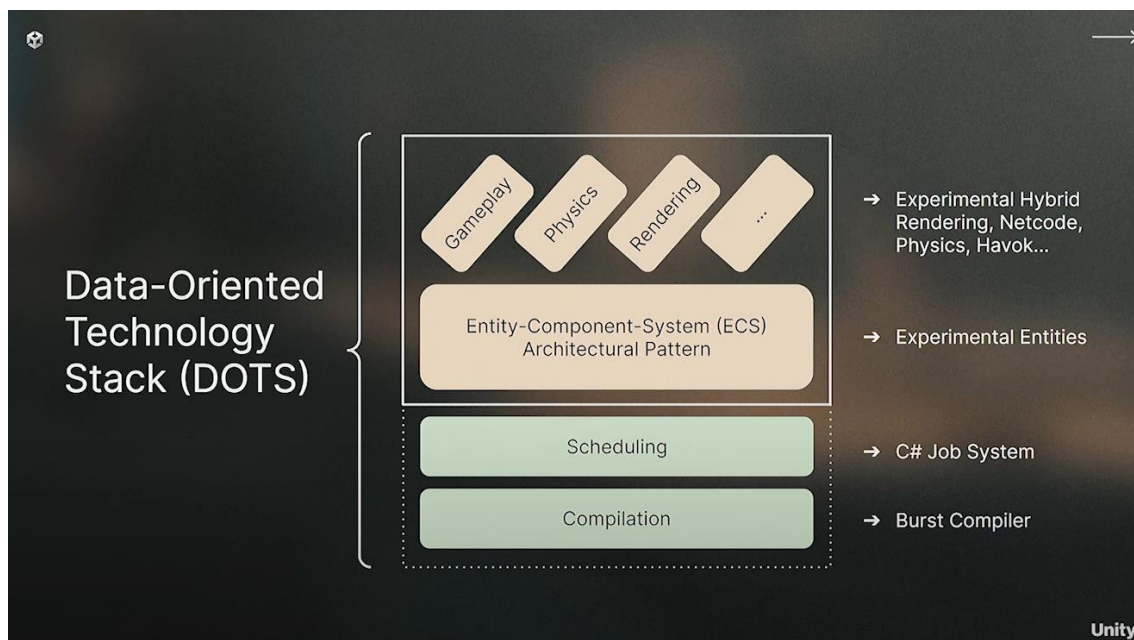


Figure 4: DOTS Framework and Packages [29]

In reality, most indie teams do not develop using ECS from the beginning [31] due to a lack of resources and an initial understanding of the system and its uses. However, as the project and team grow, the need to optimize a fully built game may arise with its problems [32].

1.3. Motivation

DOD has emerged as an alternative approach to OOD in game development, particularly within the Unity engine [33]. This research explores the motivations behind considering DOD as a potential paradigm shift and why it is relevant and exciting within the context of Unity

development. The following points outline the problematic aspects of OOD and the reasons why investigating DOD is essential:

- **Performance Optimization:**

One of the primary motivations for exploring DOD in Unity development is the need for performance optimization. With its emphasis on encapsulation and complex object hierarchies, OOD can sometimes lead to memory constraints. Game developers constantly strive for smoother frame rates, reduced loading times, and better overall performance. DOD, with its focus on data locality and cache coherence, offers the potential for improved performance and reduced memory consumption. It is essential to investigate DOD in this context to understand its potential benefits and trade-offs in performance optimization as a possible solution.

- **Scalability and Multithreading:**

As game development projects grow in complexity, scalability becomes a crucial consideration. With its heavy reliance on inheritance and polymorphism, OOD can pose challenges when distributing work across multiple threads and utilizing modern hardware architectures effectively. DOD provides a data-driven approach that lends itself naturally to parallelization and multithreading. Exploring DOD within Unity can show how it enables better scalability and concurrency management, potentially unlocking performance gains on multi-core processors.

- **Memory Management:**

Efficient memory management is paramount in game development, especially in resource-constrained environments such as mobile devices or consoles. OOD's emphasis on encapsulation and object relationships can result in fragmentation and increased memory overhead. On the other hand, DOD promotes a more direct and controllable approach to memory management, leading to improved memory utilization and reduced overhead. Investigating DOD in Unity can help identify scenarios where memory management can be optimized by leveraging data-oriented principles.

- **Architectural Flexibility:**

While widely adopted and proven effective in many scenarios, OOD may only sometimes provide the desired architectural flexibility for specific games or performance-critical systems. DOD offers an alternative mindset and methodology for structuring game code, allowing for more fine-grained control over data layout and processing. This can also be exported into the netcoding section. Unity has officially said they will not work for the current OOD netcoding packages to be performant but will focus on creating and refining the DOD net coding package [34].

- **Paradigm Exploration:**

Game development is an ever-evolving field, and exploring alternative design paradigms is essential to drive innovation and push the boundaries of what is possible. While OOD has been the dominant paradigm for game development for many years, DOD has gained traction as a viable alternative. Investigating DOD in Unity allows one to analyze its strengths and weaknesses critically, compare it against OOD, and contribute to the broader discourse on game development methodologies.

Figure 5 summarizes the general context of this research from the Data Oriented Design perspective and how it has been improved and built.

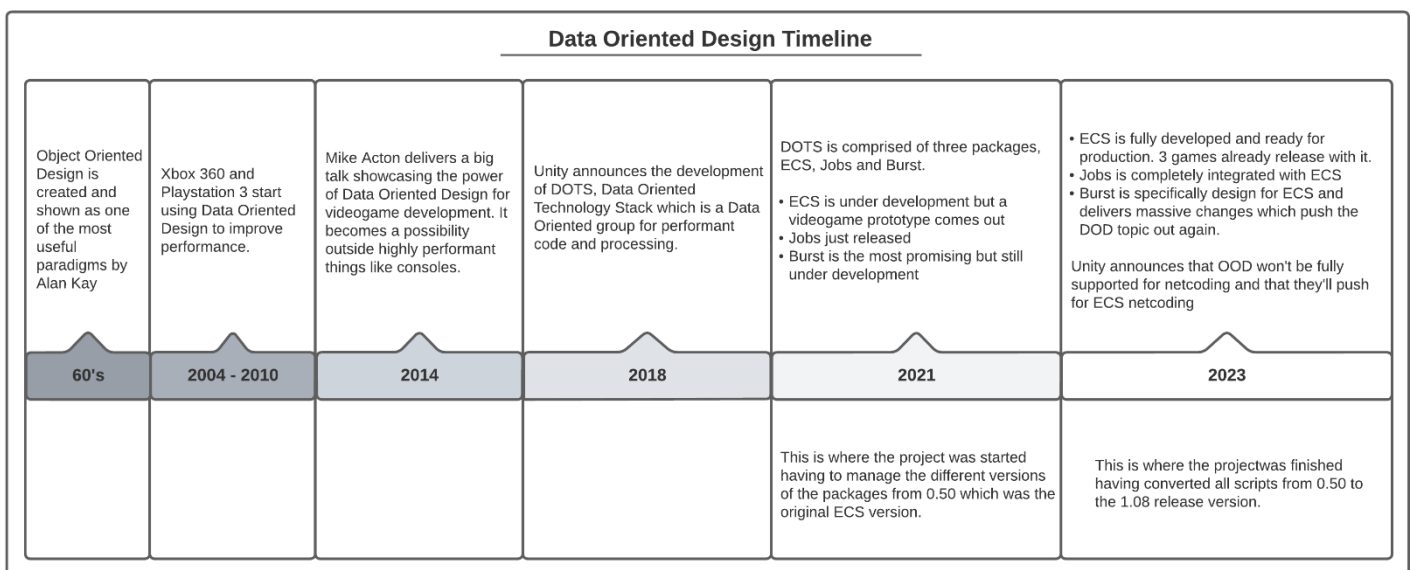


Figure 5: Summary of DOD usage over time

1.4. Industrial application: Introducing Bullet Hells

Bullet Hell games, also known as danmaku or manic shooters, have captivated players with their intense and visually striking gameplay experiences. These games belong to the *shoot 'em up* genre and are characterized by their overwhelming displays of bullets and projectiles on the screen, creating a challenging and exhilarating gameplay environment [35].

In a Bullet Hell game, players navigate their ship or character through intricate patterns of enemy bullets, dodging and weaving their way to survival. The screen becomes a symphony of projectiles, requiring precise movements and quick reflexes. The sheer volume of bullets creates a sense of tension and spectacle, demanding intense concentration and strategic positioning. This complexity is usually simplified by reducing the dimensions of movement to 2D while giving the movement impressions by scrolling, an isometric perspective, or putting the player in rails to simplify movement even more [35].



The origins of Bullet Hell games can be traced back to the early days of arcade gaming, with titles such as "Touhou Project" (**Figure 7**) and "DoDonPachi" (**Figure 6**) gaining popularity in the 1990s and early 2000s and staying popular until the present with "Touhou Project" having launched 30 games as of 2022 [38]. These games pushed the boundaries of visual effects and bullet patterns, setting the foundation for the genre's distinctive style.

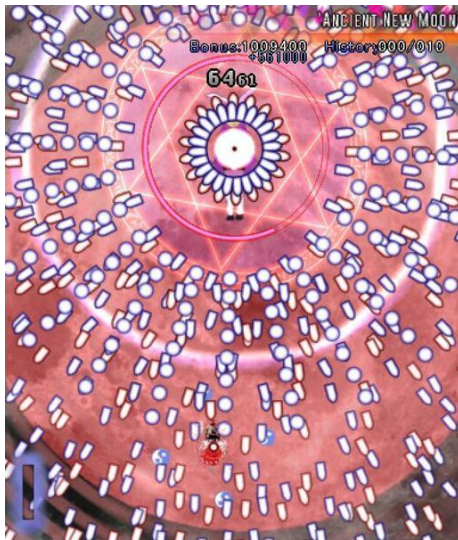


Figure 7: Touhou Project: Ancient New Moon [36]



Figure 6: DoDonPachi Screenshot [37]

One of the defining features of Bullet Hell games is their emphasis on pattern recognition and memorization. Players must learn intricate bullet patterns and enemy behaviors to devise effective strategies for survival. It is a genre that rewards practice, perseverance, and mastery, providing a deeply satisfying sense of accomplishment when navigating through a barrage of bullets unscathed.

Over the years, Bullet Hell games have evolved and diversified over the years, incorporating various themes, art styles, and gameplay mechanics. From traditional spaceship shooters to fantastical worlds with magical characters, the genre has expanded to offer players a wide range of experiences. Furthermore, Bullet Hell games have found a home on various platforms, including arcade machines, consoles, and PC, attracting a dedicated community of enthusiasts.

The popularity of Bullet Hell games stems from their unique blend of high-intensity action, visual spectacle, and strategic gameplay. These games provide an adrenaline-pumping challenge that pushes players to their limits, offering a thrilling and immersive experience that keeps them returning for more.

2. State of Art

Software design is crucial in developing modern games, providing the foundation for creating robust, maintainable, and efficient game systems. Over the years, various design methodologies and paradigms have emerged, each offering a unique approach to structuring code and managing complexity. Among these methodologies, Object-Oriented Design (OOD) and Data-Oriented Design (DOD) have garnered significant attention in the game development industry. This section explores software design's state of the art, focusing on OOD and its application in game development.

2.1. Overview of Object-Oriented Design (OOD)

Object-Oriented Design (OOD) is a popular software design paradigm that provides a structured approach to designing and organizing code. It is widely used in various domains, including game development, due to its ability to manage complexity, promote code reuse, and enhance maintainability.

The creation of OOD is not a specific moment in time, as concepts similar to objects appeared as early as the '60s. However, the first mass appearance was with the creation of Alan Kay, the Smalltalk programming language entirely based on objects [39] [40]. Since then, OOD has gained significant popularity, and lots of programming languages have developed the idea and built onto them as their core foundations [41], like Java; developed to support OOD [42] [43], C++, and C#; both created around the concepts of OOP [44] [45] [46] and more recently Python; creating as a versatile tool to support multiple paradigms [47].

The fundamental building blocks in OOD are objects, which encapsulate data (attributes) and behavior (methods). Objects are instances of classes that define their characteristic structure and behavior. OOD follows several principles to create modular and extensible systems, such as encapsulation, inheritance, and polymorphism.

Understanding OOD is essential for aspiring game developers and software engineers, as it forms the foundation for creating well-structured and maintainable codebases. By leveraging the principles of OOD, developers can design flexible and scalable systems, leading to efficient game development processes and high-quality games.

2.1.1. Definition and principles of OOD

OOD is a software design paradigm that focuses on organizing and structuring code around objects, which are instances of classes [48]. Unlike procedural or functional programming, which primarily revolves around functions and data, OOD emphasizes data encapsulation and behavior within objects. This approach brings several principles that govern OOD and provide a foundation for designing robust and flexible software systems. The core principles of OOD include the following principles:

- **Encapsulation:** Encapsulation is the principle of bundling data and the methods that operate on that data into a single unit called an object. It allows for data hiding, protecting the internal gears of an object from direct external access. For example, in a banking application, a *BankAccount* class could encapsulate attributes like balance and methods like *deposit* and *withdraw* to ensure only selected access and manipulation of the data [48] [49].
- **Inheritance:** Inheritance enables the creation of a class hierarchy, where subclasses inherit properties and behaviors from a superclass. This promotes code reuse and allows for specialization and generalization of objects. For instance, in a game development scenario, a *Character* class could serve as a superclass, and subclasses like *PlayerCharacter* and *EnemyCharacter* could inherit common attributes and methods while adding specific functionalities [48] [49] [50].
- **Polymorphism:** Polymorphism refers to the ability of objects of different classes to be treated as objects of a common superclass. This principle allows for code flexibility and modularity. A classic example is a *Shape* superclass with various subclasses like *Circle*, *Rectangle*, and *Triangle*. Even though each shape has its unique implementation of a *calculateArea* method, polymorphism allows them to be handled uniformly, simplifying code maintenance and extensibility [48] [51].
- **Abstraction:** Abstraction involves identifying and capturing the essential characteristics of an object while ignoring the irrelevant details. It allows developers to create generalized models that can be reused across different contexts. By abstracting common behaviors and attributes into base classes or interfaces, OOD facilitates code reuse and promotes a higher conceptual understanding [48] [50].

- **Association and Composition:** Association represents the relationship between objects, where one object is connected to another. Composition is a type of association that implies a stronger relationship, where an object is composed of other objects. These relationships enable the construction of complex systems by combining smaller, independent objects into larger structures. For example, in a graphical user interface (GUI), a *Window* object may have an association with multiple *Button* objects, and the composition of a *Panel* object consists of several other GUI components [48] [49] [50].

Object-Oriented Design provides a structured and modular approach to software development by adhering to these principles. Languages like UML have helped provide the tools necessary for modeling with these principles in mind making it a "standard" language to work with in conjunction [50]. It promotes code reuse, maintainability, and flexibility, enabling developers to build complex systems efficiently [52].

Besides that, Object Oriented Programming has been a topic in the White Paper on Informatics [53], which mentions it as a needed subject and one of the critical concepts for all engineers in the field.

2.1.2. Application of OOD in game development

OOD in game development is widespread due to the modular and flexible nature of OOD, which suits the complexity of games. OOD is commonly employed in the following key areas of game development:

Various entities, such as characters, enemies, items, and obstacles, play crucial roles in game development. OOD allows developers to represent these entities as objects, encapsulating their properties (e.g., position, health, speed) and behaviors (e.g., movement, interaction) within respective classes. This approach enables easier management, reusability, and extensibility of game entities. For example, a *PlayerCharacter* class, an *Enemy* class, and an *Item* class can be created in a platformer game, each with specific attributes and behaviors [54].

OOD is instrumental in implementing game mechanics, which define the rules and interactions within a game. By employing OOD, developers can represent game mechanics as a collection of interrelated objects and classes. This modular design facilitates code organization, maintainability, and the ability to iterate and expand upon game mechanics. For instance, in a puzzle game, a

Grid class, a *Tile* class, and a *Solver* class can be designed, where each class contributes to the game mechanics, such as tile movement and puzzle-solving algorithms [12].

Games often incorporate various systems, such as collision detection, input handling, and audio management. OOD allows developers to design these systems as separate, reusable components. Each system can be encapsulated within its class, ensuring clear responsibilities and promoting code modularity. For example, a game's collision detection system can be implemented using a *CollisionManager* class, which handles the detection and resolution of collisions between multiple game objects. This modular approach enables more manageable maintenance and extensibility of game systems [12] [55].

By leveraging the principles of OOD in game development, developers can create well-structured and scalable game systems. OOD promotes code organization, reusability, and maintainability, enabling efficient development and evolution of games throughout their lifecycle.

2.1.3. Advantages and limitations of OOD in game projects

OOD offers distinct advantages when applied to game projects and certain limitations developers should consider when using it.

OOD facilitates extensibility, allowing game projects to evolve and incorporate new features. Through inheritance and polymorphism, developers can extend existing classes or create new subclasses to add unique functionality without modifying the game's core structure. This flexibility is valuable in the iterative process of game development. For example, introducing a new power-up in a game can be achieved by creating a subclass of an existing *PowerUp* class, inheriting its common properties, and adding specific behavior [12].

However, OOD does have its limitations in game projects. One such limitation is the potential performance overhead associated with dynamic dispatch. While polymorphism provides flexibility but can incur a slight performance cost compared to static dispatch in languages with less dynamic features. Game developers need to consider this trade-off and optimize critical sections of the code when performance is a top priority [56] [57] [58]. Another aspect to remember is the data dispersity as we let the compiler manage it, ending with various structures over the memory.

An example is how a sphere's color may be handled in memory. Every sphere asks for its color on memory; if it is a green sphere, we set its position (**Figure 8**).

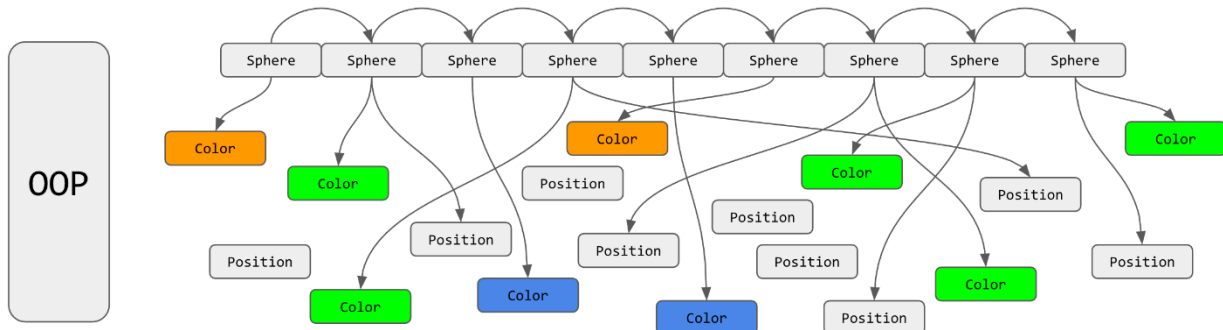


Figure 8: OOP representation of data distribution [59]

Additionally, the learning curve for developers new to OOD can pose a challenge for game projects. While acquiring a basic knowledge of OOD happens relatively fast and is taught as part of the informatic program [53], acquiring proficiency in OOD principles may take time, especially to get the patterns that will provide the best performance. Training and mentorship programs can mitigate this challenge and ensure the effective adoption of OOD practices within the game development team. However, that also takes both time and the personnel needed to teach, requiring extra effort or not focusing on it with possible future consequences [60] [61].

By considering the advantages and limitations of OOD in game projects, developers can leverage its benefits while addressing potential challenges. Properly implementing OOD principles can lead to well-structured, reusable, and maintainable codebases, enhancing the overall quality and efficiency of game development.

2.2. Introduction to Data-Oriented Design (DOD)

Data-Oriented Design (DOD) is an alternative software design approach that optimizes data structures and memory access patterns for improved performance. Unlike OOD, which emphasizes encapsulation and object relationships, DOD prioritizes efficient data representation and processing.

DOD does not have a specific creator or designer associated with it in the same way as OOD. Instead, DOD has emerged as a design approach driven by the need for performance optimization in various domains promoted by some proponents like Scott Meyers [18], Jonathan Blow [62],

Noel Llopis [63] [17], and Mike Acton [64] who has been in the Unity team to develop the ECS system [65] which we will cover in the later sections.

While DOD is not tied to any specific programming language, there are programming languages and frameworks that have grown to have an affinity for DOD principles and facilitate its implementation, like C and C++, since they are low-level languages that provide both fine-grained controls over the memory [66] [67] and SIMD-enabled (Single Instruction, Multiple Data) instructions which is a technology that allows parallel execution of the same operation on multiple data elements [68] [69] [70].

2.2.1. Definition and core principles of DOD

DOD is a paradigm that emphasizes organizing and processing data efficiently to achieve high performance. The core principles of DOD include:

- **Data Layout Optimization:** DOD advocates optimizing data layouts to improve memory access patterns. DOD aims to minimize cache misses and maximize data locality by arranging data contiguously and cache-friendly. This principle often involves using arrays of structures (AoS) or structures of arrays (SoA) representations for improved data access and parallel processing [14] [71] [72].
- **Data-Driven Design:** DOD encourages designing systems and algorithms based on the properties and characteristics of the data being processed. Instead of relying on intricate object hierarchies and polymorphism, DOD focuses on designing algorithms that operate efficiently on large data sets. This principle uses data properties to achieve performance gains, such as SIMD (Single Instruction, Multiple Data) optimizations [14] [71] [73].
- **Cache Awareness:** DOD seeks to maximize cache utilization by minimizing data dependencies and improving data access patterns. By considering the memory hierarchy and optimizing data access accordingly, DOD aims to reduce cache stalls and improve overall performance. This principle often involves batching operations and organizing data to ensure cache coherence [14] [18] [72] [74]. This improvement can have considerable consequences in process time as, for example, caching on-demand calculations on an array check (representing a map check) has an improvement by a factor of 4 while, if we could cache the total result, the improvement is by a factor of 11 (**Figure 9**).

```
i5-4430 @ 3.00GHz
Average 11.31ms [Simple, check the map]
Average 9.62ms [Partially cached query (25%)]
Average 8.77ms [Partially cached presence (50%)]
Average 3.71ms [Simple, cache presence]
Average 1.51ms [Partially cached query (95%)]
Average 0.30ms [Fully cached query]
```

Figure 10: Performance comparison depending on cache misses [14]

A similar example comes from data sorting and how having it sorted can significantly improve performance (**Figure 10**), which shows an improvement by a factor of 6.

```
i5-4430 @ 3.00GHz
Average 4.40ms [Random branching]
Average 1.15ms [Sorted branching]
Average 0.80ms [Trivial Random branching]
Average 0.76ms [Trivial Sorted branching]
```

Figure 9: Performance comparison depending on data status [14]

2.2.2. Contrasting DOD with OOD in design philosophy and approach

OOD and DOD diverge in their design philosophies and approaches. While OOD focuses on encapsulating data and behavior within objects and emphasizes the relationships between objects, DOD prioritizes efficient data representation, processing, and alignment on the cache [60] [75]

Objects are the central building blocks in OOD, and their interactions drive system behavior. OOD encourages abstraction, inheritance, and polymorphism for flexibility and code reuse. In contrast, DOD places a greater emphasis on data representation and processing efficiency. It favors a more direct and data-centric approach, optimizing data layouts and algorithms for improved performance. It is necessary for a few specific sectors where stability, performance, and lots of memory or computation power are required [76] [77].

As we saw with OOD, the example with the spheres would look drastically different in DOD, even if the bases are the same. The structure is layout in a way that lets easy access to properties like *color* and *position*, packed into buffers to reduce the number of cache misses (**Figure 11**).

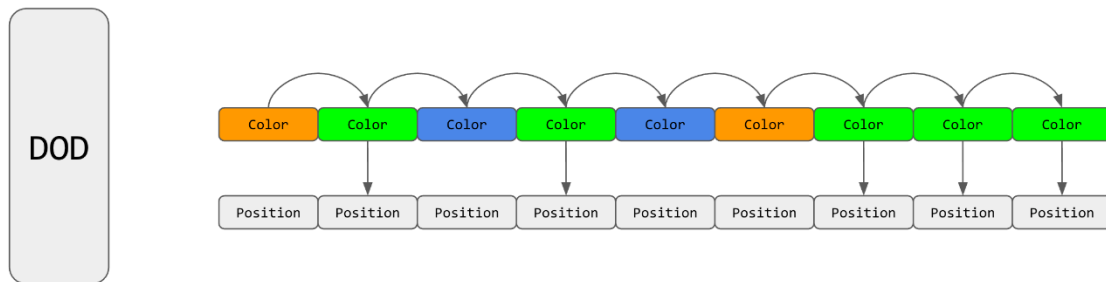


Figure 11: DOD representation of data distribution [59]

OOD encourages designing systems based on object hierarchies and encapsulated behaviors, promoting code modularity and extensibility. On the other hand, DOD focuses on the properties and characteristics of the data, designing algorithms that operate efficiently on large datasets. DOD's approach can lead to performance gains in scenarios where data processing is the primary concern, such as game physics simulations, rendering systems, and easy-to-parallelize systems [78].

2.2.3. Case studies showcasing implementations of DOD in game development

DOD has found successful implementations in various areas of game development, showcasing its performance benefits. Here are some examples and case studies:

- **Game Physics Simulation:** DOD has been effectively applied to game physics simulations, where efficient data representation and processing are critical. Physics engines can achieve significant performance gains by optimizing data layouts and leveraging SIMD optimizations. For example, the Bullet Physics library utilized DOD principles to improve collision detection and rigid body dynamics calculations [79] [80].
- **Entity-Component Systems:** DOD has been applied in the design of entity-component systems (ECS), an architecture for game development. ECS separates data and behavior, treating game entities as compositions of components. By utilizing DOD principles, ECS architectures can achieve efficient data representation and processing, improving performance. The implementation of DOD in the Unity game engine's ECS framework is an example of its successful application in game development [24] [81].
- **Cache Locality:** DOD emphasizes cache locality, which refers to organizing data to maximize its proximity to the processor and minimize cache misses (when data is unavailable and gets requested to the nearest cache memory). In modern processors, data is stored in a hierarchical cache system, with faster and smaller caches closer to the CPU [18]. Cache

locality aims to optimize data access so that frequently accessed data is stored in the closest caches, reducing the need to fetch data from slower main memory. By aligning data in memory and utilizing cache-conscious data layouts, DOD can significantly improve performance by exploiting the cache usages [74].

2.3. Comparative Analysis of OOD and DOD

In software design, OOD and DOD represent two distinct approaches, each with its principles and philosophies. This section provides a comparative analysis of OOD and DOD, examining their fundamental differences, evaluating their strengths and weaknesses in the game development context, and reviewing studies, experiments, and benchmarks that compare the two approaches.

2.3.1. Key differences between OOD and DOD from a learning point of view

OOD and DOD diverge in their core principles and design philosophies, and this leads to different learning experiences which are specific to these points:

- **Focus:** OOD focuses on encapsulating data and behaviors within objects to get the most out of inheritance and polymorphism with a "human" like approach as it is how the world is seen, which helps match the code to the visible reality for comparison and mental planning. In contrast, DOD prioritized the data representation and its abstraction outside specific contexts as objects to optimize their layouts and processing with a more abstract and not so "human" way of seeing the world, making mental planning a lot harder and pushing for tools like UML to be used to get the most accessible learning experience [14] [76] [78].
- **Granularity:** OOD usually operates at a higher level of granularity, organizing systems around objects and their relationships. On the other hand, DOD focuses on individual data elements and their processing, aiming to maximize cache utilization and improve data access patterns. This ends up creating two approaches, OOD being the creation of objects and the relationships between objects over the internal structure of each, and DOD differing with a structure focused on the operations that will be happening and the data that each will use without having an emphasis on where that data comes from or is referred by [14] [78].
- **Modifiability:** OOD's emphasis on encapsulation and abstraction can provide a high degree of modifiability and extensibility, allowing developers to add or modify behavior by working with objects and their interactions. DOD's focus on efficient data representation can enhance

performance but may require more substantial changes to the data layout when modifications are needed [14] [78].

2.3.2. Evaluation of Strengths and Weaknesses in the game development context

In the context of game development, both OOD and DOD offer distinct strengths and weaknesses:

OOD's strengths in game development lie in its ability to manage complex relationships between game entities, promote code reuse through inheritance and polymorphism, and provide high modifiability and extensibility. OOD facilitates the creation of hierarchies for game objects, supports modular development, and enables the encapsulation of behavior and data within objects. It is advantageous in scenarios that model complex interactions between entities, such as character behaviors or quest systems [60].

DOD, on the other hand, excels in scenarios where data processing performance is critical. DOD can significantly improve performance in data-intensive tasks such as physics simulations, particle systems, and AI calculations by optimizing data layouts, leveraging SIMD instructions, and focusing on cache-friendly processing. DOD's strengths lie in its ability to exploit data properties, minimize cache misses, and achieve better parallelization [14].

However, both approaches have their weaknesses. OOD's reliance on encapsulation and abstraction can introduce performance overhead, especially when dealing with fine-grained data processing or memory-intensive operations. Also, OOD's ease to use can create problems in the long term as the ease of addition and a focus on faster coding can lead to the need to reformat and document large sections of code. DOD's focus on data representation and processing may require sacrificing some of the benefits of encapsulation and abstraction, making code maintenance and modification more challenging and harder to write and create a scheme [78].

2.4. Industry Practices and Trends

The game development industry is constantly evolving, and software design practices play a crucial role in shaping the development process and the quality of games produced. This section explores the industry practices and trends related to OOD and DOD in game development, including their utilization, case studies of successful projects, and the emergence of hybrid approaches that combine elements of both OOD and DOD.

2.4.1. Use of OOD and DOD in the game development industry

OOD has been widely adopted in the game development industry for many years. Its well-established principles and practices have become the foundation for designing and implementing game systems. Game engines, such as Unity and especially Unreal Engine, heavily utilize OOD concepts to provide developers with modular frameworks and tools for building games efficiently [82].

While relatively newer in the game development industry, DOD has gained attention for its potential to optimize performance-critical aspects of game systems. Game developers are increasingly exploring DOD's benefits, particularly in physics simulations, AI processing, and rendering pipelines. DOD offers opportunities for significant performance improvements by leveraging data layouts and cache-aware processing [83].

This increase in use has started to push for UML-like structures to represent the necessary data to develop the game correctly and with structure. This has been called "Data Worksheets" and contains two tables with specific specifications, one with the data structure and one with the data management [84]. The example tables can be found in **Annex C**.

Data Structure:

- **Name:** A given name to call the variable for.
- **Type:** The design type of the variable (*int, float, vector3, quaternion...*).
- **Quantity:** The number of variables to manage (1-N).
- **Read/Write Frequency:** Number of times to read/write onto the variable. Helps with detecting bottlenecks and system structures (*Update, OnStart, OnFinish...*).
- **Reason:** Why the data is needed. It helps keep track of reasonings and dependencies in the data to leverage the order of systems and possible redundancies or structures to create.

Data Management:

- **Data:** Name the variables that need to be input into the system.
- **Output:** Name of the variables that the system will change.
- **System:** Name of the system responsible for the changes.
- **Timely:** When and how frequently the system is executed.
- **Ext. Data:** The data needed from other systems indirectly for this to work.

Creating these tables helps create a mindmap to represent in code in a simpler and more streamlined way of steps core for the ECS systems creation and management [85].

2.4.2. Case study: Unity's efforts to develop DOTS

Unity Technologies, the company behind the popular Unity game engine, has been actively exploring the benefits of DOD in game development. Their ongoing initiative, known as the Data-Oriented Technology Stack (DOTS), serves as a case study highlighting the industry's efforts to leverage DOD principles for performance optimization and scalability [86].

The DOTS framework aims to provide game developers with tools and workflows that maximize performance and leverage modern hardware architectures. By embracing DOD principles, Unity sought to enable developers to build high-performance games with efficient data processing and improved multithreading capabilities.

DOTS introduces vital concepts such as the Entity-Component System (ECS) and the Burst Compiler. The ECS architecture allows developers to design game systems (data transformations) by composing entities (a "group" of data) from smaller, self-contained components (data), promoting modularity and scalability (**Figure 12**) [81].

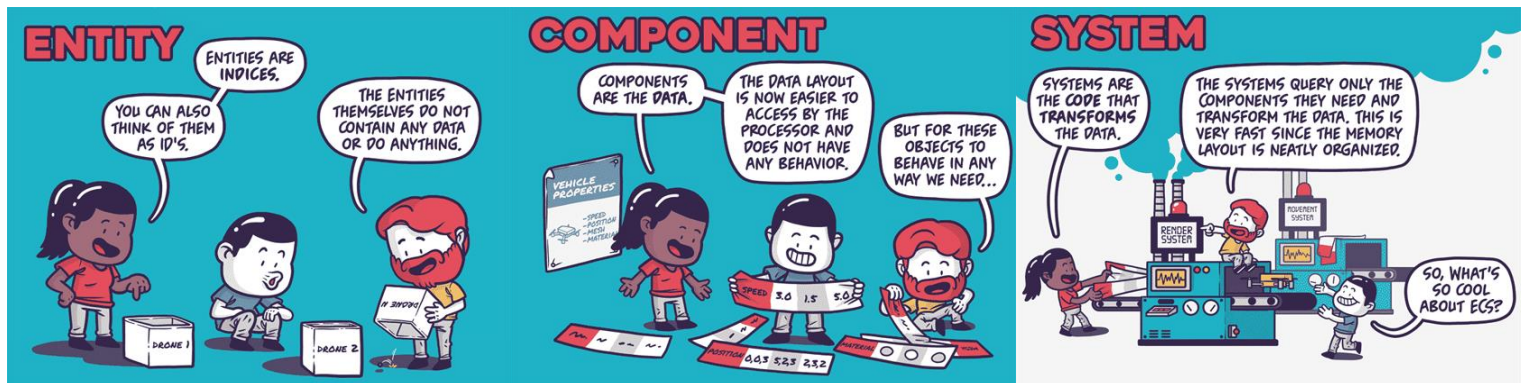


Figure 12: ECS infographic [23]

The Burst Compiler is a technology developed to optimize code for specific hardware platforms. It analyzes the code and generates highly efficient machine code tailored to the target platform's characteristics (**Figure 13**). By leveraging low-level optimizations and utilizing the capabilities of modern processors, the Burst Compiler can significantly enhance the performance of code written using the DOTS framework and the necessary help of ECS patterns [87].

The Burst Compiler works hand in hand with the ECS architecture and other DOTS components to provide developers with a powerful performance optimization tool. It allows for creating high-

performance code that takes full advantage of modern hardware advancements, such as vectorization and multithreading.

Figure 13: Burst view of assembly code [88]

Unity's efforts to develop DOTS, including the integration of Burst Compiler, showcase their commitment to harnessing the potential of DOD in game development. By providing a comprehensive framework that embraces DOD principles and incorporates advanced optimization techniques like the Burst Compiler, Unity aims to empower developers to build games that deliver optimal performance across various platforms.

While this commitment pushes for a DOD structure game, OOD is not entirely lost. Some projects leverage the DOD potential in specific parts of their games for heavy computations while remaining with a more straightforward and human-friendly structure for the rest of the project [29].

The case study of Unity's DOTS initiative demonstrates the practical implementation of DOD principles in a widely used game development engine. It serves as a testament to the industry's recognition of the performance benefits offered by DOD and its potential to shape the future of game development practices [30].

2.4.3. Comparing the learning curve of OOD and DOD

Literature highlights that the learning curve for DOD platforms may be steeper for individuals new to the concepts and paradigms associated with DOD. Developers may require more time to grasp the principles, patterns, and techniques specific to DOD and ECS. This could involve studying relevant documentation, watching tutorials, and actively applying the concepts in practical exercises [78].

However, it is worth noting that the specific learning curve can vary depending on factors such as prior experience with similar paradigms, the complexity of the platform, and the availability of learning resources. Due to OOD being a part of the curriculum in informatics degrees, the concepts underlining them are understood and applied by graduates [53]. In contrast, most DOD concepts have yet to be discovered by most, making it a significant entry barrier for programmers as grasping and using concepts that inheritably differ from their bases is not easy or fast. This is where the need for not only external tutorials, manuals, and documentation comes into play, as the concepts can be taught by other people faster than the resources can help with, just like a college subject would be [89] [90] [91].

Overall, the learning curve for both OOD and DOD approaches may involve an initial investment of time and effort to understand each design paradigm's underlying principles and tools. With practice and exposure to relevant resources, developers can gain proficiency and leverage the respective platforms effectively in game development projects.

2.4.4. Discussion of emerging trends and hybrid approaches combining OOD and DOD

In recent years, a trend has emerged where game developers combine OOD and DOD elements to leverage each approach's strengths. This hybrid approach aims to strike a balance between maintainability and performance optimization.

One common approach is to employ OOD at higher levels of abstraction to manage complex game systems while adopting DOD principles at lower levels for performance-critical components. For example, game engines may use OOD for high-level game logic and entity management while employing DOD techniques for physics simulations or heavy computational algorithms.

Another emerging trend is component-based architectures, often associated with Entity-Component Systems (ECS). These architectures combine the flexibility and modularity of OOD with the performance benefits of DOD. This has been done via a conversion workflow to convert GameObjects (objects of Unity) into Entities (used by ECS) with minimal requirements [92]. While this is standard in the current ECS system, it has heavily changed from the original conversion, making entity creation a lot more intuitive with a visual helper as an object structure which, when initialized, is immediately converted into an Entity [93].

By combining OOD and DOD in these hybrid approaches, game developers aim to balance maintainability, extensibility, and performance optimization, catering to the specific requirements of their game projects.

2.5. The current state of the Bullet Hell genre

Despite its roots in the early days of arcade gaming, the Bullet Hell genre continues to thrive and captivate players today. While it may not enjoy the same mainstream recognition as other gaming genres, Bullet Hell games maintain a dedicated and passionate following, primarily within the indie gaming community.

One of the remarkable aspects of the Bullet Hell genre is its ongoing development and innovation. Independent game developers have embraced the genre, pushing its boundaries and creating unique experiences that cater to a niche audience. These developers often infuse their games with distinctive art styles, mesmerizing bullet patterns, and intricate level designs, resulting in a diverse range of Bullet Hell games with distinct flair.

The availability of game development tools and platforms has played a significant role in the genre's continued growth. With accessible game engines like Unity and Unreal Engine, indie developers are empowered to bring their creative visions to life, including creating Bullet Hell games. However, specific engines have been built for these games through the years, like **Danmaku** [94] or its successor **Danmoku** [95], which builds upon Unity's foundations and is currently one of the most used engines with results similar to the original bullet hell games (**Figure 14**).

Several notable Bullet Hell games have garnered attention and critical acclaim recently. Titles such as "Enter the Gungeon" or "Ikaruga" have demonstrated the genre's enduring appeal and indie developers' creativity in crafting engaging gameplay experiences outside the classic *bullet*

spam. These games often combine the classic Bullet Hell mechanics with elements of roguelike, metroidvania, or narrative-driven gameplay, further diversifying the genre and attracting new players.



Figure 14: *Fantastic Poetry Festival*, created with Danmoku [95]

While the Bullet Hell genre may not garner the same mainstream attention as other popular genres, its devoted player base and continuous development showcase its enduring appeal. As indie developers push the boundaries of creativity and innovation, we expect new and exciting Bullet Hell games that captivate players with their visually stunning displays, challenging gameplay, and unique twists on the genre's conventions.

Bullet Hell games constantly push their boundaries for wilder, more extensive, and crazier ideas, making their designers seek the most optimal solutions for their designs. This has made the previously mentioned engines **Danmaku** and **Danmoku** push for new solutions like the Unity DOTS packages [94].

The need to seek better performance will be studied in this research to compare the designers' decision to try this paradigm as the bases for the engine they are building.

3. Objectives

This research project aims to compare the Object-Oriented Design (OOD) and Data-Oriented Design (DOD) paradigms in the context of game development, assess their utility and applicability to address the unique demands and constraints of modern video games, and explore the challenges and critical problems associated with transitioning between these paradigms. The project also seeks to provide insights into the performance implications of adopting OOD and DOD approaches through a benchmarking-style comparison.

Objective 1: The first objective is to compare the advantages, disadvantages, and trade-offs of OOD and DOD in game development. This comparison will provide a deeper understanding of the strengths and weaknesses of each paradigm and identify specific use cases where one approach may exhibit advantages over the other.

Objective 2: The second objective focuses on assessing the utility and applicability of OOD and DOD paradigms in addressing modern video games' unique demands and constraints. By considering factors such as lots of objects to render, complex algorithms, and modular designs, this assessment will provide insights into the practicality and effectiveness of each paradigm in meeting the requirements of modern game development.

Objective 3: Another objective is to explore the challenges and critical problems associated with transitioning from OOD to DOD or vice versa. This investigation will delve into the difficulties faced during the implementation of OOD and DOD in game development projects and provide a comprehensive understanding of the potential hurdles developers may encounter when working with either paradigm and transitioning from the standard OOD to DOD paradigms.

Objective 4: The final objective is to analyze the performance implications of adopting OOD and DOD approaches. The project will compare frame rates, memory usage, and loading times through benchmarking. This analysis will provide valuable insights into the performance trade-offs associated with each paradigm, helping developers make informed decisions when selecting an appropriate design approach.

By achieving these objectives, this research project aims to contribute to the body of knowledge in game development and expand the understanding of the DOD paradigm while providing recommendations for their appropriate usage in different game development scenarios.

4. Methodology

This research project adopts an agile scrum methodology to facilitate the systematic exploration and comparison of Object-Oriented Design (OOD) and Data-Oriented Design (DOD) in game development. The project is iterative, leveraging online meetings and email communication to ensure continuous updates and feedback.

The Scrum methodology allows flexible adaptation to evolving research requirements and encourages frequent feedback loops. The project is divided into manageable sprints, each with specific goals. During each online meeting, we discuss the project's current status, identify any challenges encountered, and make decisions regarding the next steps.

The research started with longer scrum cycles as research took more prolonged periods, just like creating the documentation and the bases to build the project, which is when the scrum cycles became shorter to manage any possible roadblocks that may occur mixing regular communication via email as a means to provide detailed updates on individual tasks and address any questions that arose during the project.

4.1. Development methodology

The development methodology employed for implementing the game using both OOD and DOD paradigms involve using the Unity game engine and its Entity Component System (ECS) framework for the DOD approach and the standard C# for the OOD implementation. The goal was to ensure a fair and comparable comparison between the two design paradigms while maintaining modularity and user-friendly interactions like setting the data in the editor without needing to go to the code for bullet speed or the number of enemies to spawn. Unity provided the necessary tools and functionalities for both OOD and DOD implementations.

The OOD implementation followed traditional object-oriented principles, emphasizing encapsulation, inheritance, and polymorphism to structure and organize game objects and their associated behaviors.

The ECS framework was utilized for the DOD approach, which focuses on the data-oriented organization and performance optimization, creating an API that handles most of the complex relationships and a clear guideline to optimize for performant assembly code with Burst.

Efforts were made to keep the comparisons between the OOD and DOD implementations as close as possible. This involved designing the data components to be implemented similarly in both paradigms, using the already available resources when necessary, and trying to showcase the strengths of both paradigms.

The OOD implementation represented game objects as classes, fundamental building blocks with well-defined relationships and behaviors. Delegates and interfaces enabled objects to adapt and react in various ways based on specific requirements or events. Delegates allowed for the encapsulation of methods, providing a flexible mechanism for callback functions and event handling like object-pool expansions.

Conversely, interfaces defined contracts that specify a set of methods that implementing classes must adhere to, facilitating polymorphism and promoting code reusability, and creating interactive behaviors that are easily interchangeable.

The OOD implementation employed pooling techniques to optimize resource utilization and reduce instantiation overhead. Object pooling involved creating a pre-allocated pool of reusable objects, which could be dynamically acquired and released as needed. This approach minimized the overhead of creating and destroying objects, improving performance and reducing memory fragmentation.

On the other hand, the DOD implementation emphasized the organization and manipulation of data for optimal performance. The relationships between data entities were the primary focus, and specific systems were designed to process and operate on this data efficiently.

One of the DOD implementation's core components was using jobs that allowed for the parallel execution of tasks, enabling efficient utilization of multiple processing cores. By dividing the workload into smaller units that could be executed concurrently, jobs facilitated improved performance and scalability.

In addition to jobs, the DOD implementation employed multiple systems. These systems were responsible for performing specific operations on the data, such as updating positions, handling collisions, or spawning entities. The systems created queries that defined sets of entities based on specific criteria, allowing them to turn their execution on or off or stop as needed selectively. This selective execution reduced unnecessary computation and improved overall efficiency.

The DOD implementation used blittable data to optimize data access and processing further. Blittable data structures can be directly translated to assembly; they are unmanaged and do not contain references. Blittable data structures are critical concepts for parallel jobs and the Burst Compiler, which massively helped the performance.

The DOD implementation aimed to maximize performance, parallelism, and data access efficiency by incorporating jobs, multiple systems with queries, and blittable data structures. These techniques allowed for scalable and optimized data processing, enabling efficient utilization of available hardware resources and enhancing overall performance in the game development context.

4.2. Evaluation methodology

This project investigates and compares OOD and DOD in game development. It seeks to provide insights into their effectiveness, performance, suitability for game development, and the human requirements to adopt and utilize them.

The comparison evaluation methodology follows Basili's Goal Question Metric (GQM) methodology [96]. It involves several steps to structure the evaluation process. Firstly, clear goals are defined, such as assessing effectiveness, performance, suitability, and human requirements of OOD and DOD. Based on these goals, specific questions are formulated, guiding the evaluation process. Relevant metrics, such as frame rate or maximum number of objects/entities, are defined to measure and answer the questions. Data is collected, aligned with the metrics, and analyzed to conclude. Then, the findings are interpreted and reported. By employing the GQM methodology, this evaluation ensures a structured and rigorous approach to assess these design approaches.

4.2.1. Research Questions

- What are the specific use cases in game development where OOD or DOD exhibits advantages over the other?
- What challenges are encountered when utilizing OOD and DOD in game development?
- What key concepts are necessary to transition from OOD to DOD successfully?
- What are the costs associated with migrating an existing project from OOD to DOD?

4.2.2. Metrics

- Comparative performance analysis in various game development scenarios using OOD and DOD.
- Identification and analysis of challenges faced during the implementation of OOD and DOD in game development.
- Assessment of the key concepts required to transition from OOD to DOD successfully.
- Evaluation of the costs of migrating an existing project from OOD to DOD.
- Examination of the advantages and disadvantages of DOD compared to OOD in game development.

4.2.3. Data Collection

This research will involve extensive literature reviews, studies, and case analyses to gather relevant data and information about OOD and DOD in game development. It will also develop a prototype with OOD and DOD as their paradigms to conduct CPU and memory usage tests, among other metrics.

4.2.4. Data Analysis

The collected data will be analyzed to provide comprehensive insights, comparisons, and recommendations regarding the advantages, challenges, costs, and overall effectiveness of OOD and DOD in game development. The analysis will be conducted on the same case scenarios for both paradigms to try and minimize differences that the different settings may have on performance. Conclusions will be drawn based on this analysis, addressing the research questions.

4.3. Methodology choice

Basili's GQM model is a structured approach used in research to establish clear goals, pose specific research questions, and identify appropriate metrics for measurement and evaluation. In this project, the GQM model provided a clear and systematic framework for conducting the experiment and analyzing the results in a purpose-driven way to connect them with the initial goals (**Figure 15**).

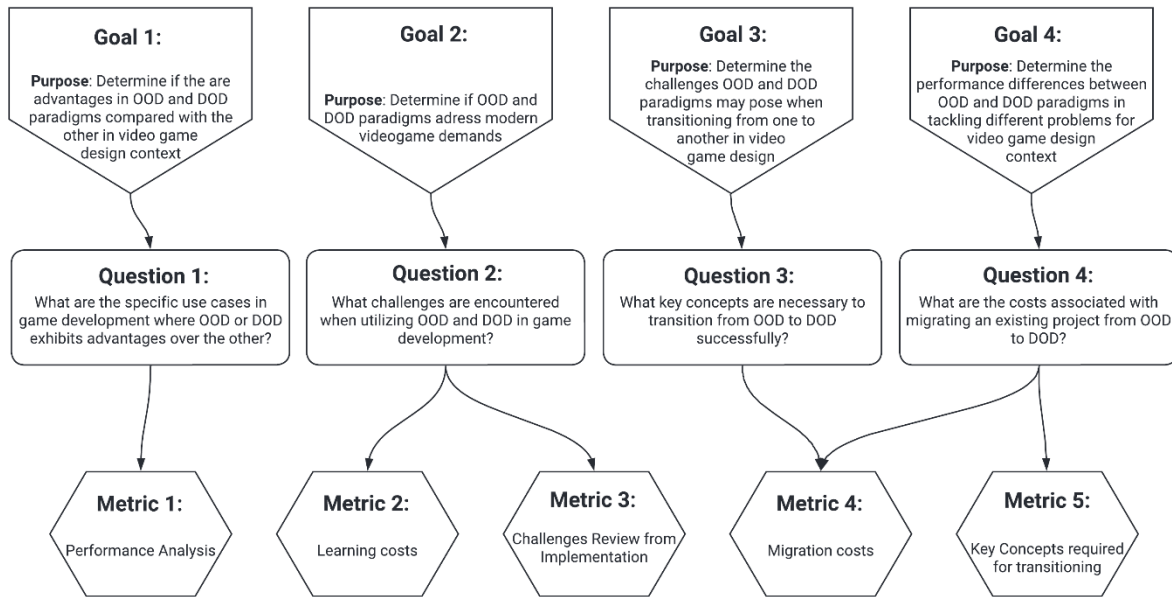


Figure 15: Scheme application of the GQM methodology

In addition to adopting the GQM model, this research project can be categorized as an in-silico experiment [97]. An in-silico experiment is a simulation-based experiment using computer models, algorithms, and virtual environments. It simulates and analyzes complex systems or processes using computational tools and techniques.

In this context, the in-silico experiment compares the OOD and DOD paradigms in video game development. The experiment utilizes computer models, simulations, and performance analysis to evaluate each paradigm's advantages, challenges, and trade-offs, except for the person-based learning curve. By experimenting in a virtual environment, researchers can control variables, gather precise data, and study the impact of design choices on game development without the need for physical prototypes or real-world testing.

Combining the GQM model and the in-silico experiment approach ensures a systematic and rigorous methodology for this research project. It allows for formulating specific research questions, selecting appropriate metrics, and exploring the OOD and DOD paradigms within a controlled virtual environment.

4.4. Working routines

This research project adopts an agile Scrum methodology to facilitate the creation of the prototype. The project follows an iterative approach, leveraging regular email communication to ensure continuous updates and progress tracking and prevent bottlenecks.

The Scrum methodology allows flexible adaptation to evolving research requirements and encourages frequent feedback loops. Due to the research nature using a technology being created in parallel, the need for constant loops is more than a requirement but a necessity. During the online meetings, we discussed the project's current status, identified any challenges encountered, and collectively made decisions regarding the next steps.

Regular communication via email serves as a means to provide detailed updates on individual tasks, share relevant resources, and address any questions or concerns that arise during the project. The in-person and online meetings are reserved for significant updates or bottlenecks needing faster resolutions or good collaboration to find solutions.

By using the agile methodology, the research can keep up with the technology changes, adapt and plan for changes that may need to be made, and try to predict and work with those changes in mind consistently.

5. Description and Implementation

The upcoming sections of this thesis will delve into the implementation and comparison of the bullet hell game prototype, which serves as a testbed for evaluating the Object Oriented Design (OOD) and Data-Oriented Design (DOD) approaches. It is worth noting that the prototype will be developed in Unity, a popular and versatile game engine known for its robust features and ease of use.

The Integrated Development Environment (IDE) chosen for this project is Rider from JetBrains to facilitate the development process and take advantage of Unity's capabilities. Rider is a powerful IDE that provides comprehensive support for Unity development, including its latest advancements, such as Data Oriented Technology Stack (DOTS) support.

Throughout the research, the game prototype will be designed and implemented with a focus on the core mechanics and gameplay elements of a bullet hell game while keeping the prototype bases not to overload the game.

5.1. Game description and mechanics

The bullet hell game prototype encompasses various gameplay mechanics, including moving, shooting, and collisions, all contributing to the gameplay experience. This section provides a detailed overview of these core mechanics while highlighting the utilization of basic graphics, in-editor parameterization, and comprehensive documentation.

- **Moving:** The player should be able to move freely along the available axes. This means having 2D movement over a plane in the prototype to create an older top-down perspective for the player.
- **Shooting:** The player should be able to shoot whenever required, specified by some parameters. The bullets should be able to be changed at runtime as well as their parameters like speed or their amount. Shooting should have some depth, like creating multiple bullets in the same wave around the player or specific attack arcs.
- **Collisions:** Collisions play a crucial role in the game prototype. Collisions should mainly occur between enemies and bullets. The enemies should lose an amount of health specified by the bullet as it gets destroyed or a specific action gets triggered.

Additionally, the prototype should offer in-editor parameterization allowing developers to adjust various gameplay parameters without modifying the underlying code. This capability enables rapid iteration, fine-tuning mechanics, and efficient experimentation during development.

Comprehensive documentation accompanies the game prototype, outlining the game mechanics and implementation details and providing guidelines for modification and expansion. This documentation supports the project's educational purposes and is valuable for knowledge sharing and future enhancements.

In the subsequent sections, we will explore the implementation of the bullet hell game prototype using the DOD and OOD approaches. Through this analysis, we aim to gain insights into the benefits and trade-offs of each design paradigm in terms of performance, maintainability, and overall game development experience.

5.2. OOD implementation

OOD promotes code modularity and reusability by breaking down the game's functionality into modular and reusable components. Each component, encapsulated within its class, encapsulates its data and behavior, allowing easy reuse and maintenance. For example, the *PlayerManager* class handles player input management, movement, and shooting, promoting modularity and separating player-related logic from other gameplay mechanics.

Encapsulation is a fundamental principle of OOD, ensuring that data and methods are grouped within a class. This promotes code organization and readability, as related functionality is contained within well-defined boundaries. Encapsulation also reduces dependencies between different game parts, making maintaining and modifying specific components easier without affecting the entire codebase. The encapsulation of enemy behavior within the *EnemyBehaviour* class exemplifies this principle, enabling centralized control and management of enemy entities while isolating their logic from other game systems. This is further explored by implementing interfaces that encapsulate specific projectile behaviors that can be interchanged and constructed easily and expanding the code without the need to understand what the underlining programs do just by "attaching" the new interface implementation.

In addition to scalability, the OOD approach simplifies code maintenance and debugging. The structured class hierarchy and encapsulation make isolating and fixing issues within specific objects or components easier. Changes or bug fixes can be made to individual classes without

impacting the rest of the system, promoting easier debugging, and reducing the risk of unintended side effects.

The OOD implementation also facilitates collaboration and teamwork among developers. The modularity and clear separation of responsibilities provided by OOD allow team members to work on different components independently, promoting parallel development with an initial setup.

5.2.1. Implementing the code

We will explore various scripts and classes developed to bring the game to life and examine the reasoning behind their implementation. These codes represent an implementation that closely resembles the mechanics and functionalities of a bullet hell game, showcasing the utilization of OOD principles to create a robust and modular system.

The implementation utilizes structs, interfaces, and encapsulation to achieve efficient data organization and improve performance. Structs, such as the *BulletParameters* struct, encapsulate the necessary parameters for spawning bullets, allowing for efficient passing and management of data. Interfaces, like *IBulletBehaviour* and *IBulletSpecifics*, define contracts that ensure consistent behavior across different bullet types and promote code modularity and flexibility.

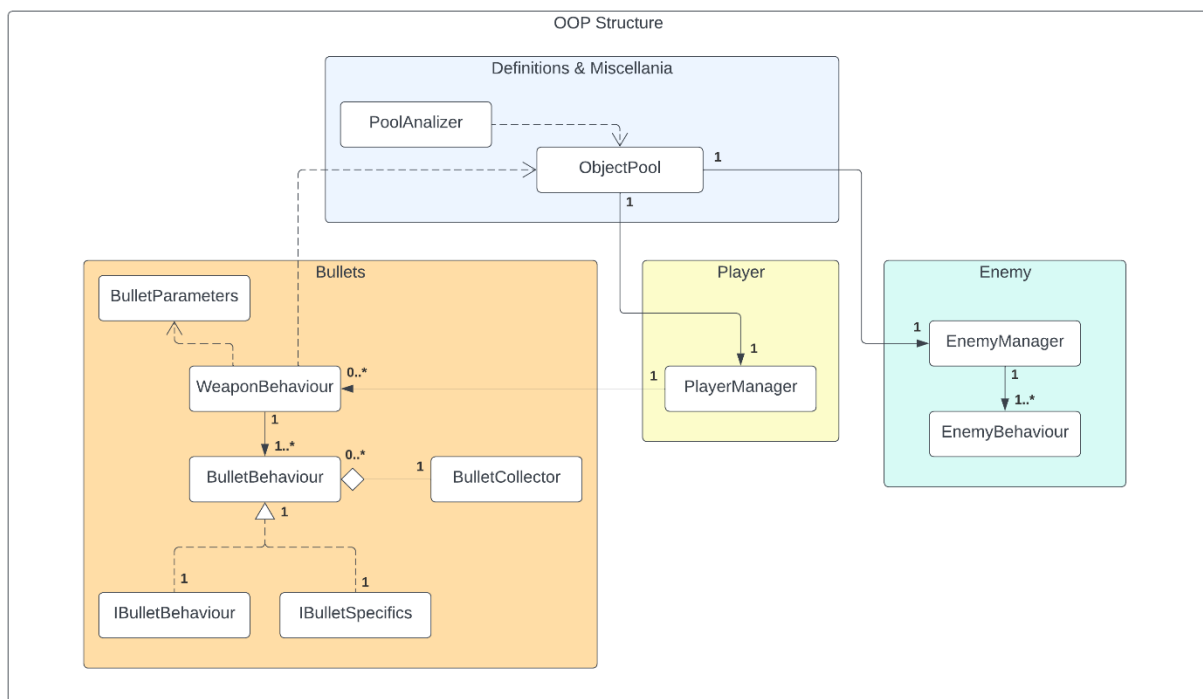


Figure 16: Structure for the Object Oriented Structure

Encapsulation helps separate concerns and isolate specific functionalities within individual scripts, contributing to better code organization and ease of maintenance.

Figure 16 shows how the structure was organized with the least connections between different classes, creating complexity within the modules and keeping the modular approach with interfaces to construct the complexity.

5.2.2. Interactions and pipeline

The game prototype utilizes various procedures to create a cohesive gameplay experience. These procedures are interconnected and communicate with each other to ensure smooth gameplay flow and proper functioning. The following outlines the interaction between these systems:

- On Start:
 - Object pools are created using the *ObjectPool* script. These pools manage the instantiation, usage, and recycling of game objects such as bullets and enemies. The game optimizes memory usage by pre-creating object pools and avoids the overhead of creating and destroying objects during gameplay.
 - Weapons are instantiated and assigned their respective bullets by the *PlayerManager* script. Each weapon type has a corresponding *WeaponBehaviour* script attached to it, which handles the behavior and properties of the associated bullets. This setup allows different weapons with unique bullet types and behaviors to be easily created, assigned, and managed by the player.
- On Update:
 - Input is checked and handled by the *PlayerManager* script. This includes capturing player movement commands and shooting actions.
 - Bullets, players, and enemies are updated. *PlayerManager* and *EnemyManager* update the positions of their managed objects, and each bullet manages itself through the specified interfaces.
 - Bullets invoke their specific implementations (interfaces), such as *IBulletBehaviour* and *IBulletSpecifics*. These interfaces define the methods for bullet update, collision detection, and destruction. Bullets exhibit unique behaviors and effects by calling the appropriate methods based on their specific bullet type.
 - The Unity physics engine handles collision detection. When collisions occur between game objects, such as bullets and enemies, Unity's collision detection system triggers appropriate collision events. The relevant scripts then process these events to handle the effects of collisions for their situations.

- On Collision:
 - When a collision occurs between an enemy and a bullet, the *BulletBehaviour* script detects the collision event. It communicates with the corresponding enemy object before handling itself for destruction or anything specified in the *IBulletSpecifics*. Towards the collided enemy, bullets may apply damage, trigger special effects, or reduce the enemy's health based on its specific implementation defined by the *IBulletSpecifics* interface. Similarly, enemies may interact with bullets upon collision, potentially reducing their lifespan or applying defensive measures.
- On Destroy:
 - When bullets or enemies are destroyed, or at the end of their lifespan, they are returned to their respective object pools using the *ObjectPool* script. This ensures efficient memory management and allows for the reuse of game objects instead of constantly instantiating and destroying them. The game maintains a consistent pool of available objects by returning objects to the pool, reducing memory overhead and improving overall performance.

These interconnected systems and their interactions create a dynamic and modular gameplay experience and ease of gameplay design.

5.2.3. Script division

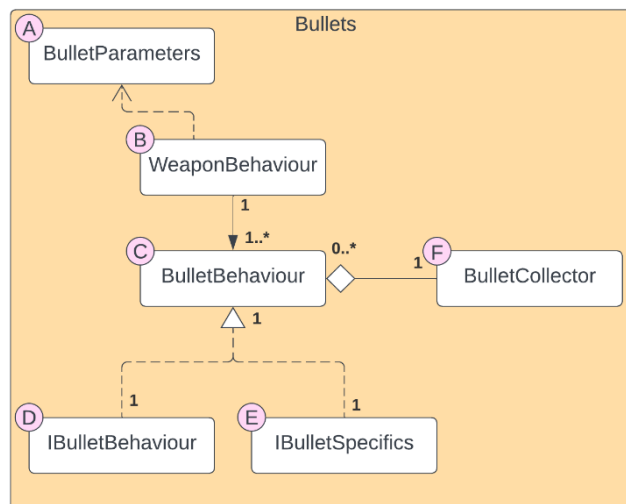


Figure 17: Bullet relationships

The bullets were the most challenging part to design and implement. They are the most modular section, as every Bullet Hell will implement many of them, requiring easy-to-use implementations that can be expanded. For this, the use of interfaces was convenient, letting future coders create a new interface implementation and then assign it via the editor. The weapon class also shows this creation, which encapsulates all the bullets and their behavior, creating self-managed sections going down into the bullet behavior.

- **BulletParameters: (Figure 17.A)**
 - Logic: The *BulletParameters* script defines a struct to encapsulate all the parameters required to spawn a bullet. Organizing the bullet-specific data into a single structure makes it easier to pass and manage the necessary information when creating bullets dynamically during gameplay.
 - Benefits: A struct allows for efficient memory allocation and performance, as structs are value types with a stack-based allocation [98]. It also promotes code clarity and organization by grouping related bullet parameters, making it easier to understand and maintain the codebase.
- **WeaponBehaviour: (Figure 17.B)**
 - Logic: The *WeaponBehaviour* script is responsible for managing the behavior of a specific type of bullet. It handles the setup of bullets, including their initial position, speed, and other parameters, based on player input and predefined values.
 - Functionality: When the player initiates a shooting action, the corresponding *WeaponBehaviour* script pools a new bullet object and assigns it the appropriate parameters and behavior according to the specified type.
- **BulletBehaviour: (Figure 17.C)**
 - Usage: The *BulletBehaviour* script controls the specific behavior of individual bullets in the game, handling their initialization, updating, collision detection, and "destruction."
 - Functionality: Each bullet instantiated in the game is associated with a *BulletBehaviour* script. When a bullet is spawned, it initializes its speed based on the predefined bullet type. During gameplay, it updates the bullet's position, applies movement patterns or transformations, and detects collisions with other game objects, such as enemies.
- **IBulletBehaviour: (Figure 17.D)**
 - Usage: The *IBulletBehaviour* interface defines the required methods for setting up and updating a bullet's behavior. A new one can be created to change the bullet's movement and selected by the editor to directly link the new movement pattern.
 - Functionality: The *IBulletBehaviour* interface is a contract that bullet-specific scripts must implement, which creates an easy-to-use and attached method to change the bullet's behaviors. It ensures all bullet types adhere to standard methods for setup and updating.
- **IBulletSpecifics: (Figure 17.E)**
 - Usage: The *IBulletSpecifics* interface defines the required methods for updating, handling collisions, and destruction of a bullet. A new one can be created to change



the bullet's destruction behavior, like having an explosion that damages nearby enemies.

- **Functionality:** The *IBulletSpecifics* interface is a contract that bullet-specific scripts must implement, which creates an easy-to-use and attached method to change the bullet's behaviors. It ensures all bullet types adhere to standard updating, colliding, and destruction methods.
- **BulletCollector: (Figure 17.F)**
 - **Reasoning:** The *BulletCollector* script manages bullets that may leave the screen's view by directly returning them to the pool. It ensures efficient memory usage and prevents unnecessary calculations on bullets that are no longer visible.
 - **Benefits:** The *BulletCollector* script helps optimize performance by removing bullets that are no longer relevant, reducing the computational load on the game. By efficiently managing the lifecycle of bullets and removing unnecessary objects from memory, it contributes to a smoother gameplay experience and better overall performance.

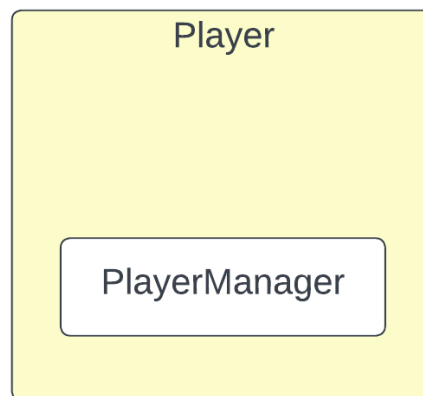


Figure 18: Player relationships

The player was the easiest to implement as it is the typical player controller, which implementation is standard in the industry and has been seen along the different subjects in the career. It reads from the player and creates the weapons, making it the most exciting part as it reads from the editor's input to generate them, creating a simple and interactive UI dynamically.

- **PlayerManager: (Figure 18)**
 - **Logic:** The *PlayerManager* script handles input management for the player, including movement and shooting. It also orchestrates the creation of different weapons, given some start parameters.
 - **Benefits:** Encapsulating the player-related functionality within the *PlayerManager* script promotes modularity and separation of concerns. It allows for a clear separation between player input handling and other gameplay mechanics, making the codebase easier to extend or modify.

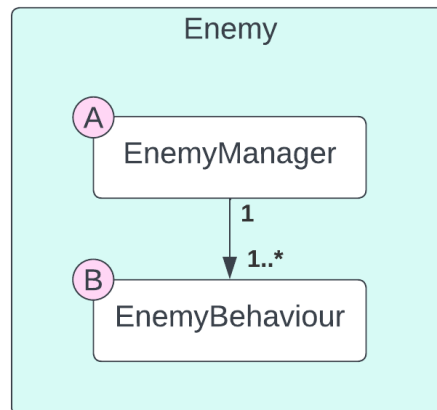


Figure 19: *Enemy relationships*

Implementing the enemies was relatively straightforward since they follow the same structure as the player, except for movement management and the ability to have multiple instances. Their design is modular, so creating more complex behaviors by incorporating the bullet's approach with the enemy interfaces is easy. While the prototype did not require this level of complexity, it was still an option for future development.

EnemyManager: (Figure 19.A)

- Logic: The *EnemyManager* script sets and pools all the enemy entities in the game. It handles the enemy waves.
- Benefits: Encapsulating enemy wave functionality within the *EnemyManager* script allows for centralized control and management of enemies' waves, simplifying the creation of wave types and structs to parametrize them specifically in complex ways [99].

• **EnemyBehaviour: (Figure 19.B)**

- Usage: The *EnemyBehaviour* script controls the behavior of enemy entities in the game, including their setup, updating, collision detection, and destruction.
- Functionality: Each enemy entity in the game is associated with an *EnemyBehaviour* script. When an enemy is spawned, it initializes its movement patterns for the update cycle. During gameplay, it updates the enemy's position towards the player, applies movement patterns or behaviors if required, detects collisions with other game objects, and handles the "destruction" of the enemy when appropriate, such as when its health is depleted.

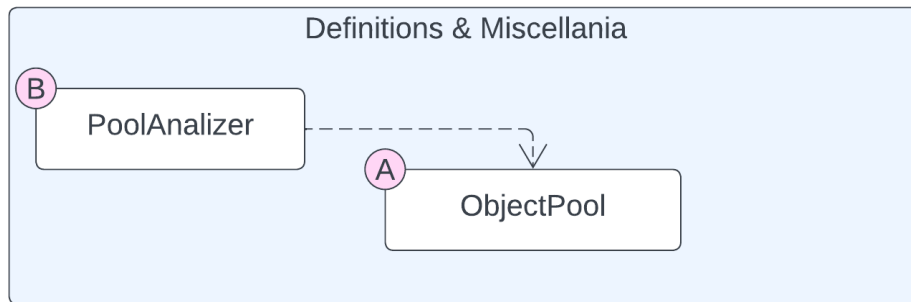


Figure 20: Definition & Generic relationships

Creating a struct to manage pools was the prototype's first step. While Unity has its pooling system, creating one from scratch gave more flexibility towards letting editor variables define specific behaviors and understanding the insides of a rather complex design pattern. This helps compare it with the DOD implementation as I got a hands-on experience with the complexity of pooling and managed objects.

- **ObjectPool: (Figure 20.A)**
 - Logic: The *ObjectPool* script manages the creation, usage, and expansion of object pools in the game, such as bullets and enemies. It optimizes performance by recycling and reusing objects instead of instantiating and destroying them repeatedly during gameplay.
 - Benefits: Object pooling offers several advantages, including reduced memory allocation and deallocation overhead, an improved performance due to minimizing expensive instantiation and destruction operations, and better control over the number of active objects at any given time. Using an object pool promotes efficient memory usage and smoother gameplay performance.
- **PoolAnalyzer: (Figure 20.B)**
 - Usage: The *PoolAnalyzer* script provides a user interface (UI) that displays real-time information about the status of active bullets and enemies in the game.
 - Functionality: The *PoolAnalyzer* script continuously monitors the bullet and enemy pools, retrieving the current number of active objects. This information is then displayed in the UI, visually representing the object pool usage. The *PoolAnalyzer* script is a debugging tool and provides insights into using the object pooling system.

5.2.4. In-Editor view

The development of the bullet hell prototype in Unity benefits from the in-editor view, which provides a convenient and intuitive environment for script management and customization. The *Enemy*, *Player*, and *Bullet* scripts can be accessed and modified within the Unity editor. This allows developers to fine-tune gameplay mechanics, adjust parameters, and iterate on the game's design more efficiently. The hierarchy tries to express and convey that by their names, as shown in **Figure 21**. Here are the details of the essential fields that can be directly edited within the editor:

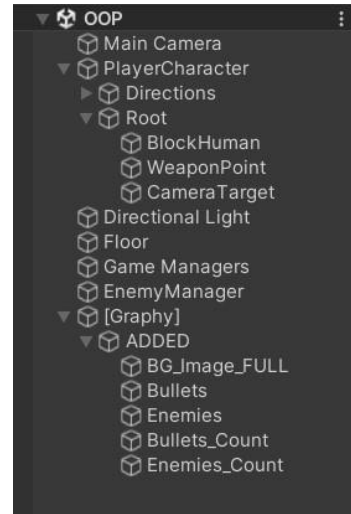


Figure 21: OOD Hierarchy view

- **Enemy Script: (Figure 22)**

- The Unity editor displays the *Enemy* script as a component attached to enemy entities or prefabs. Developers can access and modify various properties of the enemy, such as *movement speed*, *spawn radius*, or *health*.
- The only requirement is the *Player Position* as a reference since the default movement of the enemies is towards the player, and having a reference is less expensive than searching it at runtime.

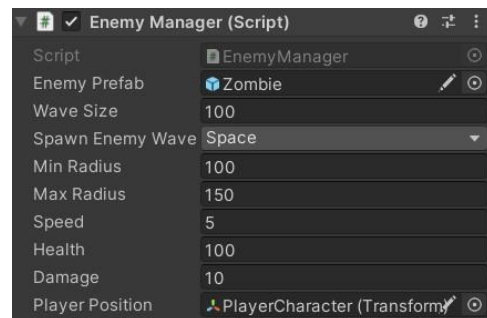


Figure 22: Enemy manager parameters

- **Player Script: (Figure 23)**

- The *Player* script is also visible as a component within the Unity editor, attached to the player character. Developers can access and configure different parameters associated with the player, including *movement speed*, *initial weapon*, and *modes*.
- There are more requirements in this as we need to know what will be moving (root)

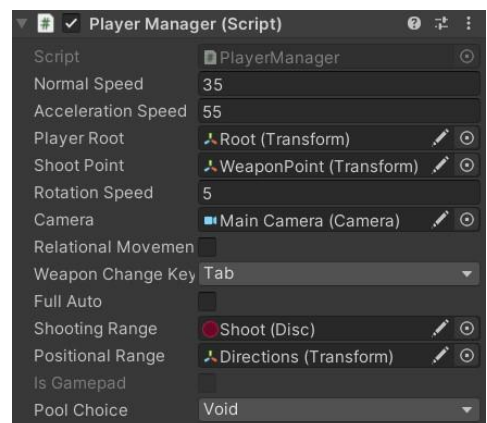


Figure 23: Player manager parameters

and where the bullets will be shoot from (shoot point). Like with the *Enemy*, we store the camera reference to avoid getting it at runtime. Both *Shooting* and *Positional Range* are visual details that can be entirely omitted.

- **Bullet Script: (Figure 24)**

- The *Bullet* definition is accessible from the editor as well, with some specific setup. Thanks to Odin's plugin, *List<>* can be shown in the editor directly [100]. This lets the programmer create multiple types of bullets without going to the code side.
- The bullets are defined in two sections: pool and bullet settings. The pool settings are related to the overall management, the bullet prefab, the pooling amount, and how it is expanded when needed, among other parameters which tune the overall functionality. On the bullet side, everything can be changed on how the bullet works, *speed*, *arc to shoot*, *amount*, and even specific behavior with the interfaces directly from the editor.
- Modularity has a cost in the code, but Unity does a wonderful job connecting them with as much ease as possible, making coding these modular classes much more effortless.

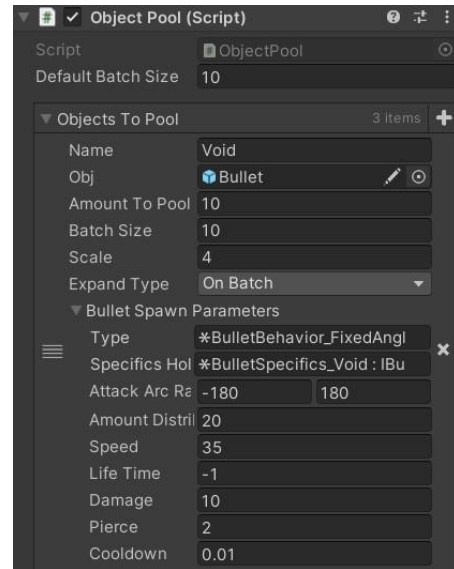


Figure 24: Bullet parameters

5.3. DOD implementation

5.3.1. Understanding the framework

The Unity framework for DOD is called ECS by the sections that define it, Entities, Components, and Systems. Throughout this section, these terms will be constantly used among some related ones, so first is understanding these concepts. A general explanation can be seen in **Figure 25**, but let us dive into the specific insights for each part:

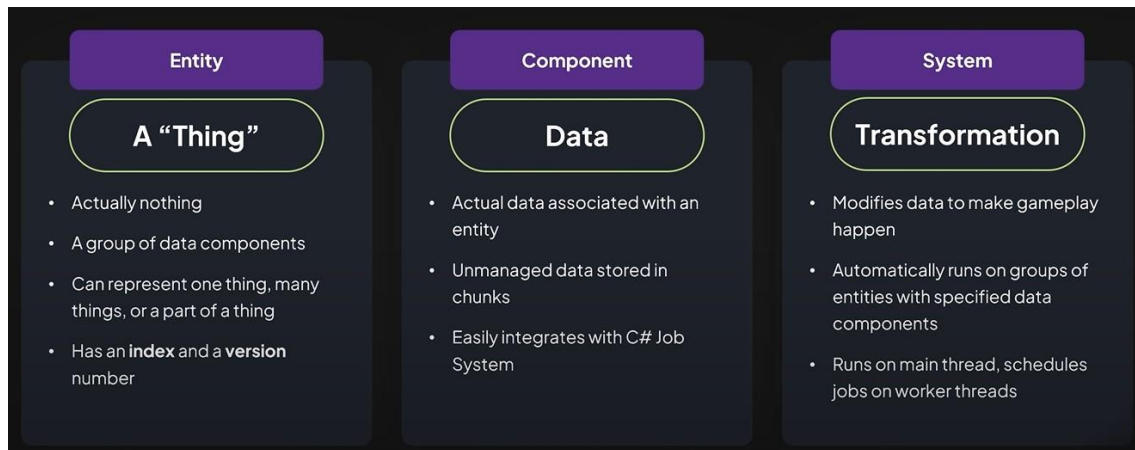


Figure 25: General definition of ECS [101]

- **World:** (Scene container) A collection of entities that can interact with each other. It can be seen as a classic Unity scene but for code only.
- **Entities:** (A "thing") Entities represent the fundamental building blocks of the game world. They are unique identifiers that encapsulate and group related data components. In itself, it is nothing but can represent lots of things as data groups. It has an index and version number to identify them in the respective chunk and index (**Figure 26**).



Figure 26: Internal definition of an entity [102]

Entities then are ordered by archetypes depending on the components they have. This ensures that all the arrays of data are shared among all the entities in the chunk (**Figure 28, Figure 27**). This also means that creating and destroying entities is simple; just adding the data components to the specific archetype arrays and future systems will pick them up and process them. This makes the use of pooling unnecessary, as enabling and disabling entities is not a possible thing. Only components can be enabled, a recent feature that comes with a performance cost [103].

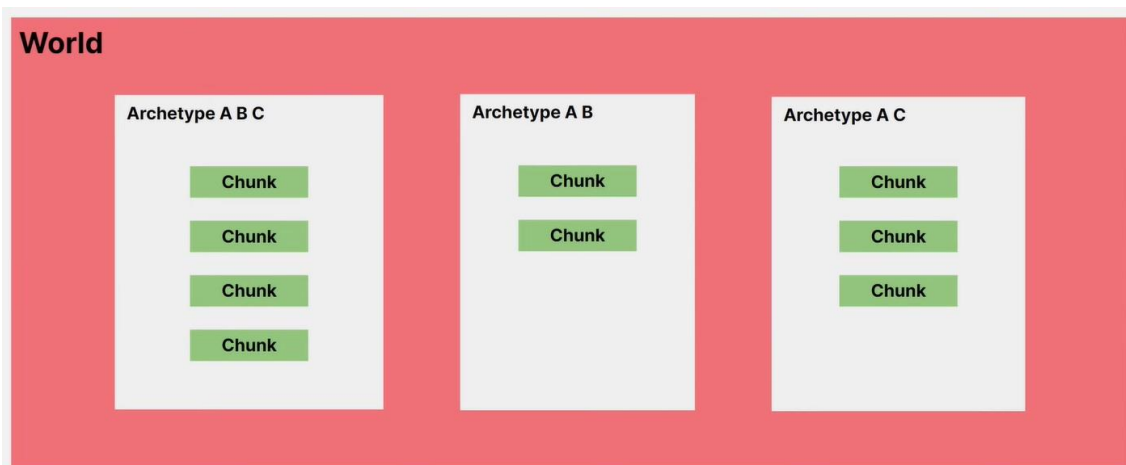


Figure 28: Archetype definition inside a world [102]

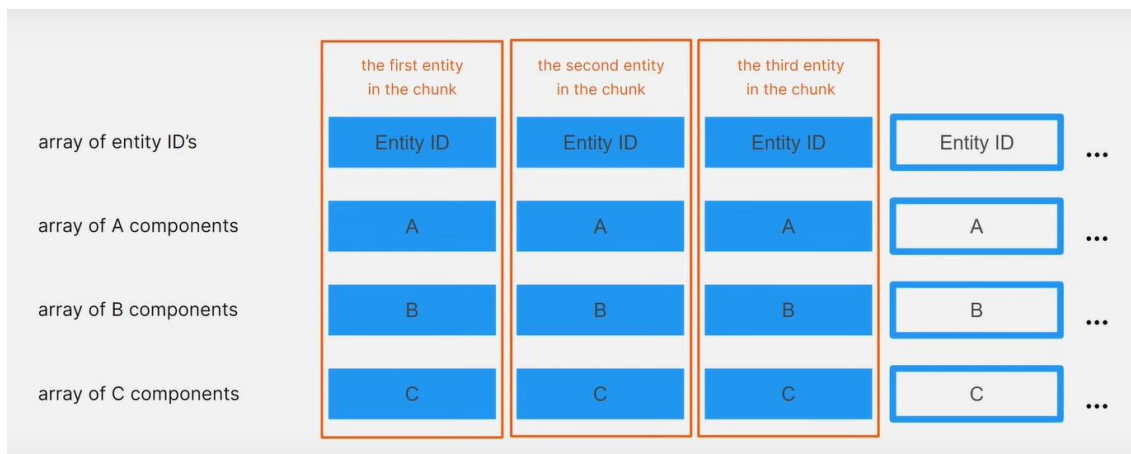


Figure 27: Entity grouping [102]

- **Components:** (Data) Components are data containers that hold specific attributes and properties of entities. They are self-contained and devoid of any behavior. The stored data is stored in chunks and mostly unmanaged so that we can know the size in memory, and it does not contain references managed by the garbage collector.
- **Systems:** (Transformations) Systems modify and operate on entities with specific components. They contain the logic and behavior that manipulate the data stored in components to create the gameplay. They run on the main thread and usually schedule their complexity into jobs that act on the worker threads.
- **Jobs:** (Thread) Jobs are responsible for making code run in parallel along threads or workers, as Unity calls them. They can run in different types, per entity, per chunk, and transform... Leading to multiple options to tackle specific problems.
- **Burst Compiler:** (Assembly and SIMD) The Burst compiler is a crucial component of the Unity Jobs system, specifically designed to optimize C# code for performance. It leverages low-level optimizations and generates highly efficient machine code tailored for specific platforms. It has specifications like the inability to handle managed data types,

enums, and other things shown in the documentation [104] [105]. Its uses can be used outside ECS, but the strict requirements make it more challenging.

5.3.2. Creating the data worksheet

As explained in the State of Art section and **Annex C**, DOD has a specific type of diagram called Data Worksheet. It helps create easy-to-use and understandable systems to manage specific data types and see their order and dependencies. Here are the tables of the Data Worksheet for this project. Due to the prototype's low number of entity types, bullets (**Table 1**), players (**Table 2**), and enemies (**Table 3**) get an initial table.

Bullets (BulletTag)	Type	Read Frequency	Write Frequency	Why do we need it?
Transform	LocalTransform	Once every frame	Once every frame	Rendering and movement
Velocity	float	Once every frame	Authoring	Movement
Direction	float2	Once every frame	Once on init	Movement
Collision Radius	float	Once every frame	Authoring	Collision calculations
Pierce	int	Once every frame	Authoring On Collision	Destruction of the bullet

Table 1: Data worksheet for bullet entities

Player (PlayerTag)	Type	Read Frequency	Write Frequency	Why do we need it?
Transform	LocalTransform	Once every frame	Once every frame	Rendering and movement
Velocity	float	Once every frame	Authoring	Movement
Rotation Velocity	float	Once every frame	Authoring	Movement

Table 2: Data worksheet for player entities

Enemies (EnemyTag)	Type	Read Frequency	Write Frequency	Why do we need it?
Transform	LocalTransform	Once every frame	Once every frame	Rendering and movement
Velocity	float	Once every frame	Authoring	Movement
Health	int	Once every frame	Authoring Once on init	Destruction of the enemy

Table 3: Data worksheet for enemy entities

After defining the unmanaged data for the entities, we need to define how they will be transformed and used. For that purpose, we use the second table of the Data Worksheet for data transformations (**Table 4**). This table reads, "*The **System/Job** uses the **Input** to generate changes in **Output** depending on **Other Data**.*"

Input	Output	System/Jobs	When and how frequently?	Need other data?
Transform Velocity Rotation Vel. Direction	Transform	BulletMovementSystem EnemyMovementSystem PlayerMovementSystem	Once every frame	-
Health EnemyTag	Health	EnemyMovementSystem	Once every frame	Transform BulletTag Collision Radius
Transform PlayerTag	Direction Entity	BulletSpawnSystem	On-demand by player	Transform PlayerTag
Health	Entity	EnemySpawnSystem	On-demand by player	Transform PlayerTag
Transform Pierce Collision Radius BulletTag	Pierce Entity	BulletPierceCheck	Once every frame When pierce<0, the bullet is destroyed	Transform EnemyTag
Health EnemyTag	Entity	EnemyHealthCheck	Once every frame When health<0, the enemy is destroyed	-
Transform BulletTag	Entity	DestroyWhenOffCamera	Once every frame When the bullet is off- camera, it is destroyed	-

Table 4: Data worksheet component transformations

5.3.3. Implementing the code

With the data layout defined, we need to define the actual code to implement them. One of the key factors of ECS is to separate data from behavior. Since the data was previously defined on the Data Worksheet, this section will delve into the core algorithms that transform those components.

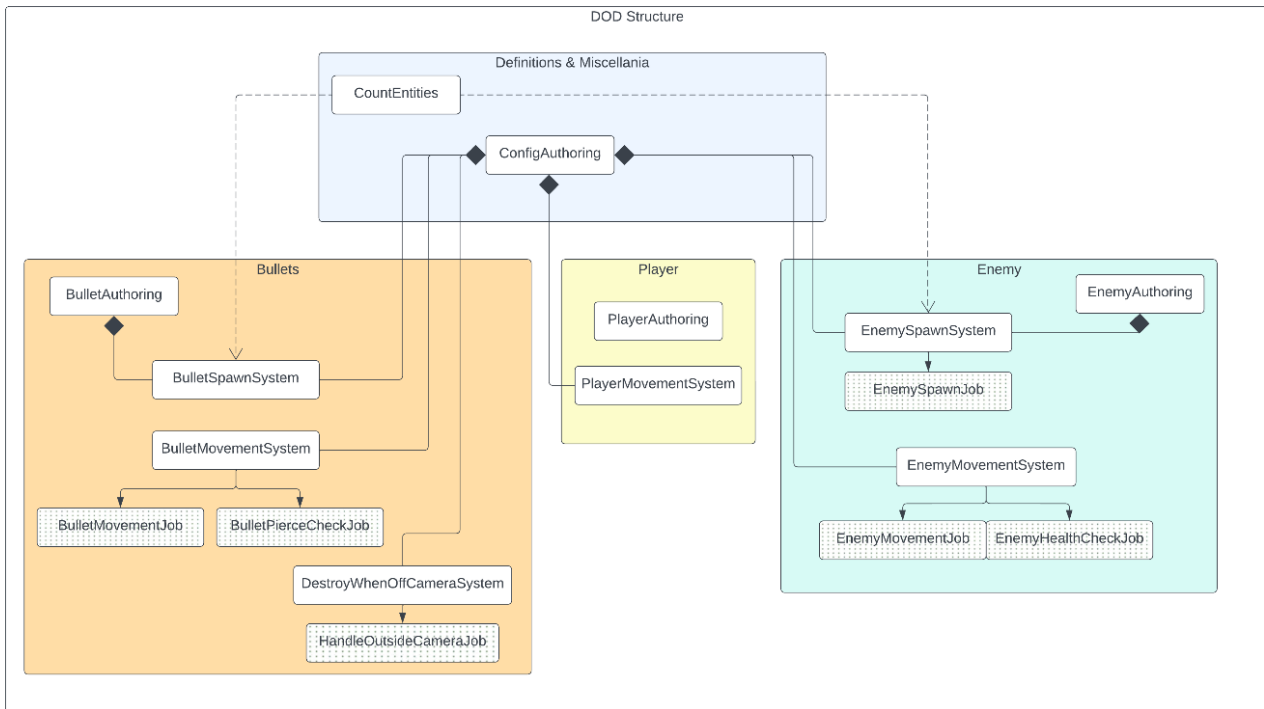


Figure 29: Structure for the Data Oriented Structure

Figure 29 shows how the structure was organized. The approach is somewhat similar to the OOD one, with some key differences. Authoring scripts like *BulletAuthoring* or *EnemyAuthoring* are data components that define the entity's initial components (chunk) to add when it gets instantiated, just like a prefab would, like the bullet's pierce in the *BulletAuthoring* or the enemy's health in the *EnemyAuthoring*. Apart from the authoring, most systems will schedule jobs to do the computation, not to overload the main thread. Those jobs are marked with a dotted background in **Figure 29**.

5.3.4. Pipeline

When the game is launched, a World is created where all the entities will be created. Then the managed pipeline starts:

- On Awake
 - Any subscene in the present Unity scene will be treated as inside the created world and will be converted to entities. This is the case of the Player, which lives inside the

- ECS subscene. It has an Authoring component that tells the system how the entity should be created and exposes any parameters that may be needed onto the editor.
- Systems are created and ordered randomly among their groups keeping any specified order if needed.
 - Data chunks are created for the current entities and are allocated in memory.
 - On Start
 - Systems check their start conditions. These are preconditions for the script to run, usually related to a singleton existing or an entity being created. They also set their dependencies, the input and output data from **Table 4**. They will be set accordingly, meaning that if data is read or written only, it will be expressly specified to enable better parallel work on the job threads. Specifying the relationship with the data is the programmer's responsibility; if not specified, the code will default to read and write, creating a less streamlined and performant code.
 - On Update
 - All systems that are enabled and meet their preconditions run, executing their codes and setting any jobs they need to run. They also update their dependencies if needed and set them for the next frame to create an action pipeline.
 - Jobs run in parallel along all threads as they are activated. They cannot be communicated with and will only return information when it fully finishes being unable to be stopped until then. They should know their dependencies, and the programmer should give them the correct ones to avoid inaccurate data. This means that a job should depend on other jobs being completed creating a tree structure along all systems like in **Figure 30**, where job D depends on jobs E, F, and G being completed before it runs, which means that in **Figure 30**, status, the job D would still need to wait for jobs G and H to finish.

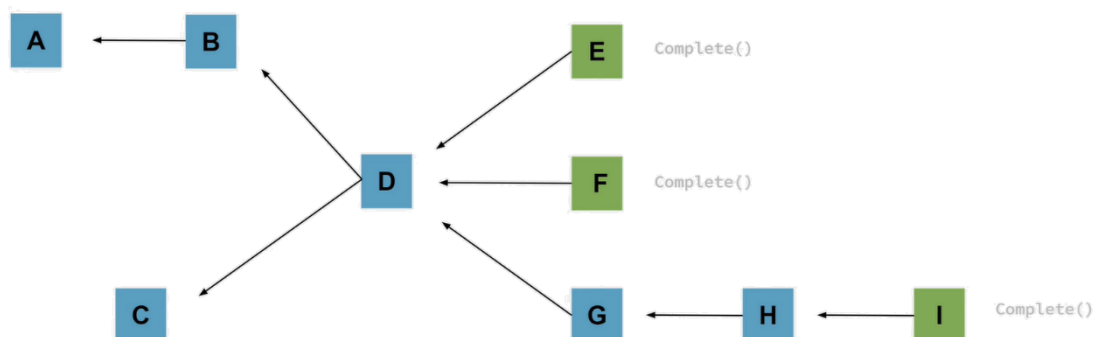


Figure 30: Job branching dependencies [106]

- On Frame End
 - This step is remarkable as it is unique to the framework. It runs when all systems have finished their code, and any internal job has also finished. Then, it sees the next frame's system dependencies and checks if any job is locking the required data. If it is, it will be called to be complete before the next frame starts. While this may look like it can create bottlenecks, the Burst Compiler makes jobs super performant, usually making the main thread the one to end the latest.
 - Another part of this step is that any in-memory operation is done at this step, so it does not affect any systems. This includes instantiating and destroying entities.
- On Frame Start
 - This is also a remarkable step as it is unique to the framework. When the frame has ended and all the dependent jobs have finished, all systems check their data chunks to see if there are entities in them activating or deactivating their update sections accordingly and adding any dependencies that may arise from the activation.

Knowing how the pipeline of information works lets us build the systems accordingly, which will be shown in the following section.

5.3.5. Script division

In the DOD implementation, we have organized the scripts into different categories based on their functionality and purpose. This division allows for better modularization and separation of concerns, enabling efficient data processing and optimization. Here is an overview of the script division in our DOD implementation:

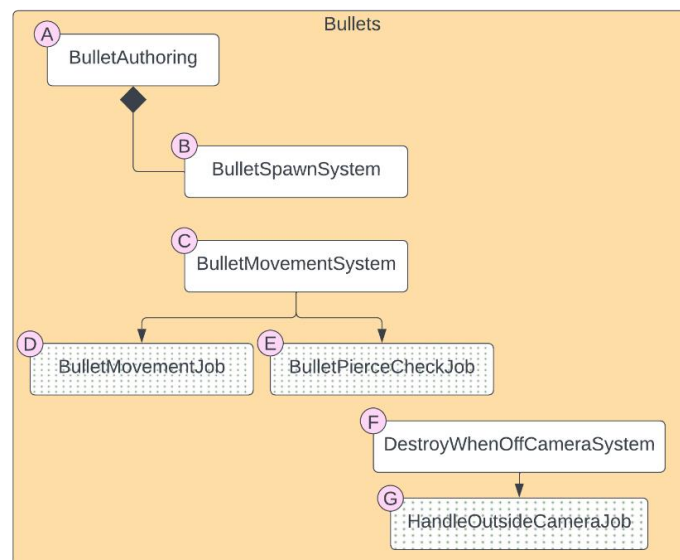


Figure 31: Bullets processing relationship



Managing the bullets in DOD was surprisingly more straightforward than the OOD toughness and modularity. The tag system makes it much easier to change the bullet's behaviors, and just changing one for another lets another system process it, changing its behavior. Collisions were done by hand as the official Unity Physics was not released when the project was being developed [107].

- **BulletAuthoring: (Figure 31.A)**

- Logic: The *BulletAuthoring* script handles the conversion aspects of bullets, allowing designers to define their components in the Unity Editor. It provides an interface for configuring bullet characteristics such as *Pierce* and sets the components the bullet entities will have when created as a template or archetype to follow.
- Benefits: A particular conversion method encapsulates the exact data we require in a script, only needing minor changes to update the components, which is as simple as using *AddComponent<>* with a new one in the conversion (**Figure 32**).

```
public override void Bake(BulletAuthoring authoring)
{
    var entity = GetEntity(TransformUsageFlags.Dynamic);

    AddComponent<BulletTag>(entity);
    AddComponent<Velocity>(entity);
    AddComponent<Pierce>(entity, new Pierce {Value = authoring.pierceCount});
}
```

Figure 32: Bullet authoring script

- **BulletSpawnSystem: (Figure 31.B)**

- Functionality: The *BulletSpawnSystem* is responsible for spawning bullets in the game world based on predefined criteria. It handles the instantiation and initialization of bullet entities. This was done inside the system instead of in a job to evaluate the differences it would create when running.
- Benefits: By utilizing a system dedicated to bullet spawning, we can efficiently manage bullet creation and ensure consistent behavior. This allows for easy customization of spawning rules and dynamic bullet generation during gameplay.

- **BulletMovementSystem: (Figure 31.C)**

- Functionality: The *BulletMovementSystem* manages the movement of bullets within the game world and their current state by creating two jobs that handle that.
- **BulletMovementJob: (Figure 31.D)**

Functionality: The *BulletMovementJob* is a parallel job system that performs the movement calculations for bullets, leveraging the power of multi-threading and

data-oriented processing. This system is run per-entity basis, so every bullet gets called by the system to be updated.

Benefits: We can create multiple behaviors with multiple jobs along different systems using a job system if we have multiple bullet types. This is different from the OOD approach as it cannot be created modularly since we have to have predefined data chunks.

- **BulletPierceCheckJob: (Figure 31.E)**

Functionality: The *BulletPierceCheckJob* is a job system that checks for bullet collision. It detects collisions with enemy entities and determines whether a bullet should keep living (pierce=>0) or be destroyed (pierce<0). This is not done directly, as destroying entities is a memory change operation, so it gets scheduled to do it safely at the end of the frame.

Benefits: Using a job system for bullet collision and pierce checks allows for efficient and parallel processing along all entities. While now the Unity Physics package could now do this more efficiently, the job performance is not too different.

- **DestroyedWhenOffCamera: (Figure 31.F)**

- **HandleOutsideCameraJob: (Figure 31.G)**

Functionality: The *HandleOutsideCameraJob* is a job system that detects and destroys bullets or entities that have moved outside the camera's view and destroys them. It helps maintain performance and optimizes resource usage by removing objects that are no longer visible.

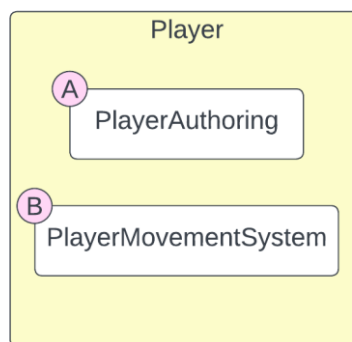


Figure 33: Player processing relationship

Managing the player is just as easy as in OOD. Implementing the Input system as unmanaged code lets the use of standard code inside the ECS system, making the transition a 1 to 1 mirror of the original code.

- **PlayerAuthoring: (Figure 33.A)**
 - Logic: The *PlayerAuthoring* script handles the authoring aspects of the player entity, allowing designers to define components specific to the player's character. It also created the template or archetype of the player.
 - Benefits: A particular conversion method encapsulates the exact data we require in a script, only needing minor changes to add new components to add complexity or new behaviors to the entity.
- **PlayerMovementSystem: (Figure 33.B)**
 - Functionality: The *PlayerMovementSystem* manages the movement logic and behavior of the player entity. It processes user input and updates the player's position and rotation accordingly.
 - Benefits: We can efficiently handle player input and movement calculations by dedicating a system to player movement. This enables specific control and the ease of adding new features like sprinting directly onto the system without rethinking the data layout.

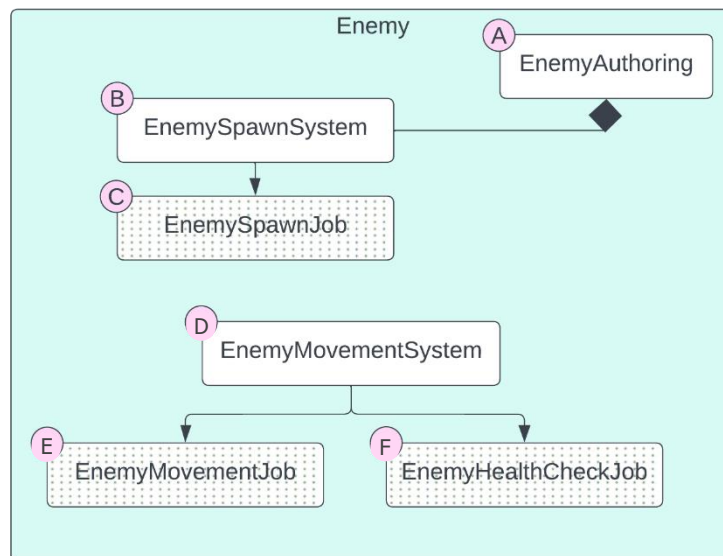


Figure 34: Enemy processing relationship

Creating the enemies was just as easy as creating the player character. The only difference is that the enemies were created at runtime, making the need to set the prefab correctly. The collision system was manually created, leading to bullets checking collisions with enemies and enemies checking collisions with bullets doing the work twice. This happens as there is no proper way of managing entities from the components without creating overheads, making it more optimal to search in one, free the Transform data and then repeat in the other.

- **EnemyAuthoring: (Figure 34.A)**
 - Logic: The *EnemyAuthoring* script handles the conversion of enemy entities, allowing designers to define attributes and properties specific to enemy characters.
 - Benefits: A particular conversion method encapsulates the exact data we require in a script, only needing minor changes to create multiple enemies that could be linked with bullets if needed. This adds complexity and needs to be manually handled by the developers to avoid losing performance.
- **EnemySpawnSystem: (Figure 34.B)**
 - Functionality: The *EnemySpawnSystem* spawns enemy entities based on predefined criteria. It utilizes a job system to parallelize and optimize enemy spawn calculations.
 - **EnemySpawnJob: (Figure 34.C)**

Functionality: The *EnemySpawnJob* is a parallel job system performing enemy spawn calculations, like the starting position to be in a radius from the player. It handles the instantiation and initialization of enemy entities.
- **EnemyMovementSystem: (Figure 34.D)**
 - Functionality: The *EnemyMovementSystem* manages enemy entities' movement logic and behavior. It updates the position and rotation of enemies based on the player's position to move toward them.
 - **EnemyMovementJob: (Figure 34.E)**

Functionality: The *EnemyMovementJob* is a job system that performs the movement calculations for enemy entities in parallel, utilizing multi-threading for optimized performance. It also checks if there are any collisions after moving, updating the health value if needed.

Benefits: Having a specific job for them means adding complexity is easy to find and add, like having a minimum radius to chase the player. It only needs a distance calculation and a data value for the radius passed to the system.
 - **EnemyHealthCheckJob: (Figure 34.F)**

Functionality: The *EnemyHealthCheckJob* is a job system that checks the health status of enemy entities and handles their destruction when necessary.



Figure 35: Definition relationship

The configuration acts as a singleton definition for all the entities setting one singleton of data for the player, enemy, and bullets to use. It also defines the prefabs and could specify more data that should be used across the systems. CountEntities is not an ECS system but a MonoBehaviour that accesses the World to request information, in this case, the entity count. I expected this to be rather hard, but the API made it extraordinarily straightforward, and only took 20 minutes to set everything up.

- **ConfigAuthoring: (Figure 35.A)**
 - Logic: The *ConfigAuthoring* script creates the game's general configuration settings and parameters via singletons of data exposed in the editor, as shown in the **5.3.6** section. It provides a centralized location for defining global game settings.
 - Benefits: By centralizing the configuration settings, designers can easily add, remove or update specific data for the entirety of the system's ecosystem from a single point.
- **CountEntities: (Figure 35.B)**
 - Functionality: The *CountEntities* script is a utility script that counts the number of entities in the game world. It provides information about the current entity count for monitoring and debugging purposes.
 - Remarks: The *CountEntities* script is a classic Unity script (MonoBehaviour) that does not live in the ECS World. It can still access a small manager to return a query of entities, and while it cannot interact with them, it can read the length of the query in this case.

These script divisions and their respective logic, benefits, and functionality demonstrate the organization and utilization of DOD principles in our game implementation. By separating scripts based on their responsibilities and leveraging the power of systems, jobs, and authoring components, we can achieve efficient data processing, improved performance, and maintainable code structure.

5.3.6. In-Editor view

The development of the bullet hell prototype in Unity still benefits from the in-editor view, just like with OOD, which provides a convenient and intuitive environment for script management and customization. The *Enemy*, *Player*, and *Bullet* entities now are containers of data and do not have any parametrization that is not per-entity managed like the pierce of a bullet which is individually changing for each bullet entity (**Figure 36**).

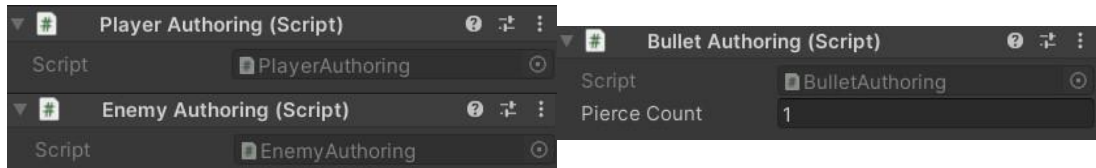


Figure 36: Authoring components

This leaves the containers with the minimum needed information and lets the main configuration handle the complexity they can individually have with all the parameters and references needed (**Figure 37**). While most of the serialized parameters are blittable data, some references prefabs. When the Authoring component gets executed, these prefabs are converted to entities, stripping any references and extra data it may have by default or creating a cloned entity of the given *GameObject* if requested, creating a twin entity that will share some components like its *Transform*.

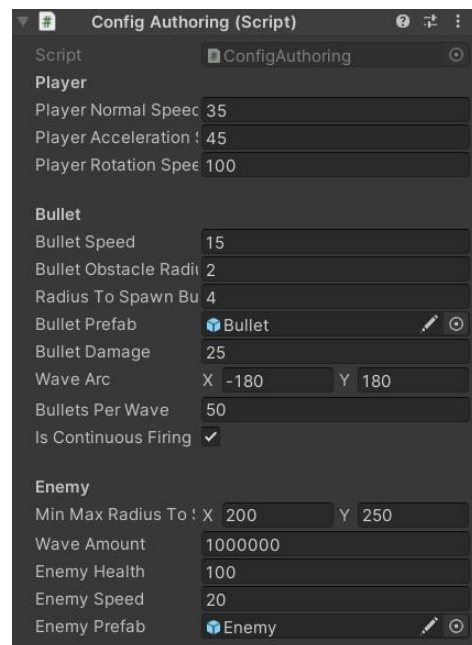


Figure 37: Configuration authoring

It is much less convoluted along the hierarchy (**Figure 38**) as it only requires data containers that are then processed into the natural hierarchy, which only exists at runtime (**Figure 39**).

Figure 39 shows the Entity Hierarchy, which shows how each container's conversion ended up at runtime (marked in yellow on the left). Two blue entities (the hexagon shape icons) represent the Bullet and Enemy Prefabs, a particular type in ECS, as they cannot be used as Bullet or Enemy. However, they are just archetype templates to copy from when required. Apart from them,

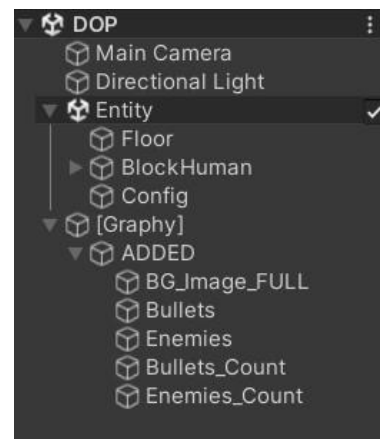


Figure 38: Basic Unity hierarchy

we can see a trail of different Unity.X where X is a system that Unity already defines for us and some other systems like the Scene.DOP.EnemyMovementSystem is one of the systems defined for the prototype's purpose directly. They act as singletons of the systems, holding the queries they will manage and any data that may be intrinsic to them.

Other important menus the ECS package allows are the Components, Archetypes, and System windows, which help see how the internal conversions and processes have worked.

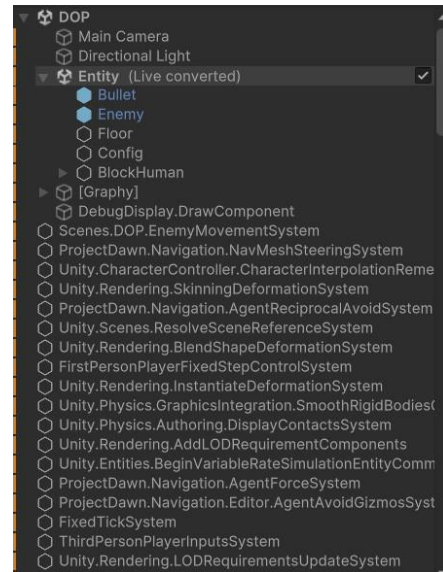


Figure 39: Entities Hierarchy window

The Components window lets us see all the components currently in the project (merging all the World) and shows their internal type and other helpful information like the size, alignment, or component type. Figure 40 shows the Pierce component, an int saved in 4B of alignment and size.

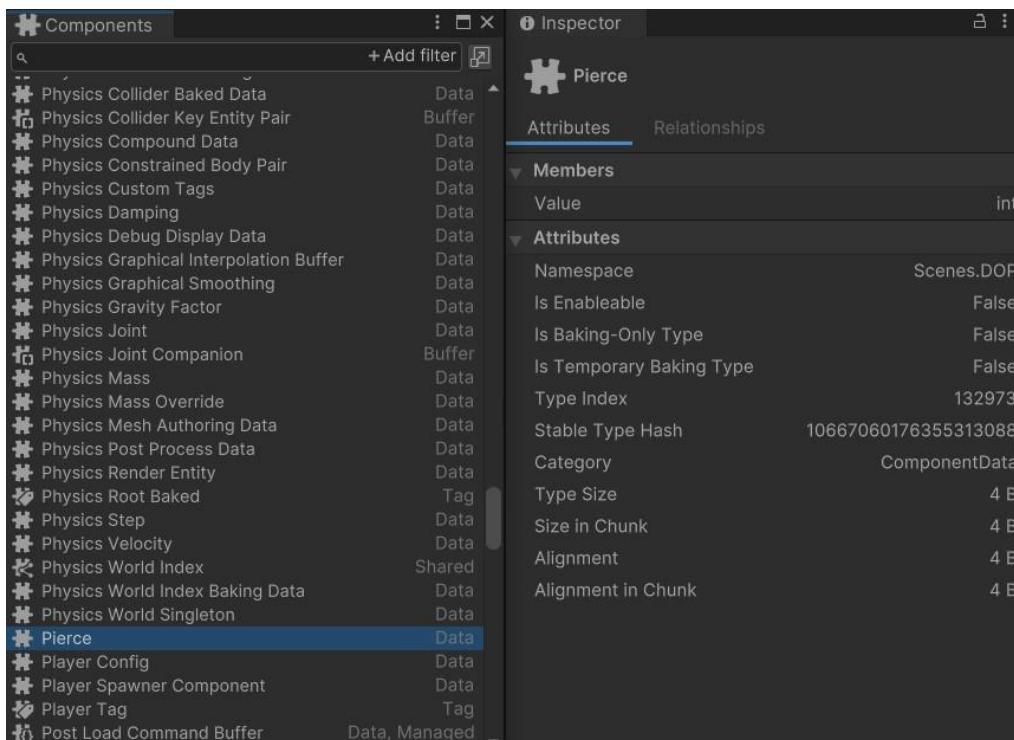


Figure 40: Components window and Pierce inspector

Like the Component window, Archetype shows the components of an entity group, as shown in **Figure 28**. This window shows the memory side letting the user see the weight of their entity in the chunk as it is a limiting factor. **Figure 41** shows the Bullet Archetype, which weighs 772B in memory, making only 20 bullets available per chunk (16K size). Most of the components have been added by Unity to render the entity correctly. However, there are some specific optimizations available to define, like the TransformUsageFlags, which let the programmer choose how the entity will be created between None (no need for Transform), Renderable (needs to be rendered but will not be moving so only LocalToWorld) or Dynamic (needs to be rendered and moved so both LocalTransform and LocalToWorld) among other options [108].

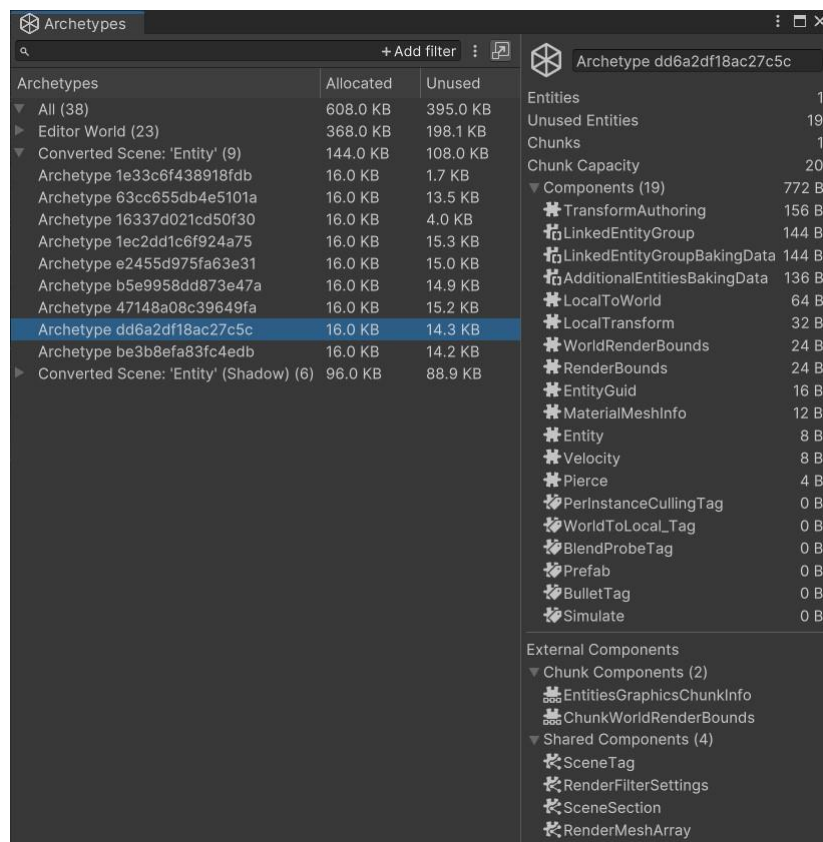


Figure 41: Archetype window and Bullet inspector

Lastly is the Systems window, which shows the running systems, their data queries, and any internal data they may have while executing every frame (**Figure 42**). Most systems are managed by Unity and are grouped into different sections to order them. The user-created systems can subscribe to those groups so they are updated together just like they can specify to be updated before or after other systems.

Systems	World	Namespace	Entity Count	Time (ms)
Initialization				
Initialization System Group	Editor World	Unity.Entities		0,00
Begin Initialization Entity Command Buffer System	Editor World	Unity.Entities	1	0,00
Retain Blob Asset System	Editor World	Unity.Entities		-
World Update Allocator Reset System	Editor World	Unity.Entities		0,00
Live Conversion Editor System Group	Editor World	Unity.Scenes.Editor		0,00
Editor Sub Scene Live Conversion System	Editor World	Unity.Scenes.Editor		0,00
Scene System Group	Editor World	Unity.Scenes		0,00
Weak Asset Reference Loading System	Editor World	Unity.Scenes		-
Scene System	Editor World	Unity.Scenes	1	0,00
Resolve Scene Reference System	Editor World	Unity.Scenes		0,00
Scene Section Streaming System	Editor World	Unity.Scenes	6	0,00
End Initialization Entity Command Buffer System	Editor World	Unity.Entities	1	0,00
Update				
Simulation System Group	Editor World	Unity.Entities		0,00
Begin Simulation Entity Command Buffer System	Editor World	Unity.Entities	1	0,00
Fixed Step Simulation System Group	Editor World	Unity.Entities		0,00
Begin Fixed Step Simulation Entity Command Buff...	Editor World	Unity.Entities	1	0,00
End Fixed Step Simulation Entity Command Buffer...	Editor World	Unity.Entities	1	0,00
Variable Rate Simulation System Group	Editor World	Unity.Entities		0,00
Companion Game Object Update System	Editor World	Unity.Entities		0,00
Transform System Group	Editor World	Unity.Transforms		0,00
Physics Display Debug Group	Editor World	Unity.Physics.Auth...		0,00
Companion Game Object Update Transform Syst...	Editor World	Unity.Entities		0,00
Late Simulation System Group	Editor World	Unity.Entities		0,00
End Simulation Entity Command Buffer System	Editor World	Unity.Entities	1	0,00
Pre Late Update				
Presentation System Group	Editor World	Unity.Entities		0,00
Begin Presentation Entity Command Buffer System	Editor World	Unity.Entities	1	0,00
Hybrid Light Baking Data System	Editor World	Unity.Rendering		0,00
Register Materials And Meshes System	Editor World	Unity.Rendering		0,00
Deformations In Presentation	Editor World	Unity.Rendering		0,00
Push Mesh Data System	Editor World	Unity.Rendering		-
Instantiate Deformation System	Editor World	Unity.Rendering		-
Push Blend Weight System	Editor World	Unity.Rendering		-
Blend Shape Deformation System	Editor World	Unity.Rendering		-
Push Skin Matrix System	Editor World	Unity.Rendering		-
Skinning Deformation System	Editor World	Unity.Rendering		-
Structural Change Presentation System Group	Editor World	Unity.Rendering		0,00
Add LOD Requirement Components	Editor World	Unity.Rendering		0,00
Update Hybrid Chunks Structure	Editor World	Unity.Rendering	3	0,00
Manage SH Properties System	Editor World	Unity.Rendering	5	0,00
Add World And Chunk Render Bounds	Editor World	Unity.Rendering		0,00
Update Presentation System Group	Editor World	Unity.Rendering		0,00
LOD Requirements Update System	Editor World	Unity.Rendering		-
Light Probe Update System	Editor World	Unity.Rendering		0,00
Render Bounds Update System	Editor World	Unity.Rendering		0,00
Entities Graphics System	Editor World	Unity.Rendering	11	0,00
Matrix Previous System	Editor World	Unity.Rendering		0,00

Figure 42: System window

Together, these tools help identify problems and memory specifications to improve and create better data layouts for the entities. Due to the packages being released on June 2023 [109], it still has bugs and problems to solve and implement. However, it also has a very active community, and developers are involved in making ECS something that genuinely shapes the future of video games and is more widely used by the community [110].

6. Economic Study

In addition to exploring the technical aspects and performance implications of OOD and DOD in game development, this research project aims to conduct an economic study to evaluate the costs associated with adopting these design paradigms. One of the key factors influencing the economic aspects is the number of hours invested in implementing OOD and DOD approaches.

Quantifying the time spent on each design paradigm makes it possible to estimate the labor costs incurred during the development process. Additionally, considering the mean salary of a programmer, the economic study will further analyze the financial implications of implementing OOD and DOD in game development, shedding light on the potential economic advantages or drawbacks of each approach.

The datasets of the time invested in each script, paradigm, and section can be seen in **Appendix A**, and a visual showcase of the data can be seen in **Annex D**.

6.1. Cost analysis

The comparative analysis of the time invested in both paradigms shows that OOD and DOD had similar times to be developed. However, it was not fully developed as the project was built in OOD and then transformed into DOD.

I have measured the time investments in OOD and DOD for the project. The OOD approach required 984 minutes (16 hours and 24 minutes), while the transition to DOD took 837 minutes (13 hours and 57 minutes). These time measurements form the basis for estimating the labor costs associated with each approach.

Estimating the costs based on the standard salaries of video game programmers provides further insights into the economic implications. Let us consider a mean salary of €15 per hour for video game programmers as an estimate from the Spanish market [111] [112] [113]. Using this value, we can calculate the estimated labor costs for implementing the project, as shown in **Table 5**.

\	€ Cost	Hours Invested	Cost per hour
OOD	246	16,4	15
DOD	209,25	13,95	15

Table 5: Costs of the project by hours

Recognizing that these cost estimates only reflect the labor costs associated with the given time investments is important. The transformation of an OOD project to DOD would involve additional efforts that should be considered, such as understanding DOD principles, refactoring code, and optimizing data layout. These factors may contribute to increased development time and potential additional costs. These additional costs also depend on the project size. In contrast, a smaller project may be easier to transform, they may not need the DOD benefits as much as a more extensive project, but they will have more work to do to transform their data layouts and script relationships. One key of the transformation is the algorithms; they stay the same from OOD to DOD, which eases one of the conversion problems.

Considering that, and as discussed in the **Future Work** section, it would be interesting to see how a more extensive project is transformed and how one is created from the ground up like the OOD was to properly compare if this data can be extrapolated to them too.

Regarding the time required to learn the DOD paradigm and its framework, we can estimate, from creating this prototype, that it would take around 120 hours to learn about this paradigm. This is a skewed value as I had prior knowledge of the subject. For someone completely new to the paradigm who wants to learn and build, it could take between 90 and 180 hours to become literate enough to develop in Unity's framework. To fully understand the underlying concepts of DOD, it could take around 300 hours.

Learning the DOD framework is no different from learning OOD in a career. OOD is studied from the first year and appropriately taught in the second year [53], with students learning by doing over time. By the end of their career, they have a deep on-field knowledge of it. The 300 hours needed to learn DOD would be equivalent to two 6 ECTS subjects [114], similar to the time dedicated explicitly to OOD throughout a career.

This lack of university knowledge is extrapolated to another challenge; finding programmers accustomed to the DOD framework or Unity's ECS framework [101]. It makes it challenging to find someone to build this type of project from the start or have someone to teach the team. However, it is a valuable skill that may become more in demand shortly. While the mindset required for this framework may not be for everyone, it is an exciting knowledge to have and to then apply to classic OOD systems and also to stand out from many other programmers.

7. Results

7.1. Prototype review

As part of this research project comparing Data-Oriented Design (DOD) and Object-Oriented Design (OOD) in Unity game development, I created a prototype to demonstrate the practical implementation and performance differences between the two paradigms. The prototype is a characteristic idiosyncrasy of bullet hell game bases. It features a top-down orthographic perspective where enemies surround the player as it tries to shoot them away to open paths (**Figure 43**).

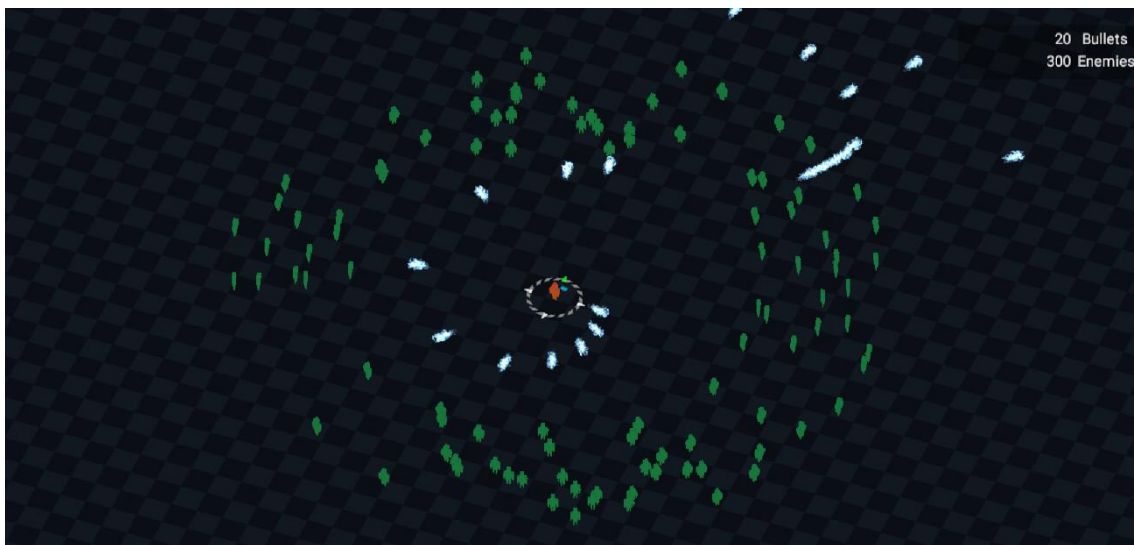


Figure 43: OOD Prototype look

Let us look at the prototype and provide a general review of its design and performance.

The prototype was a double one, each using a different paradigm, OOD and DOD, to compare their effectiveness and trade-offs. The OOD version followed traditional object-oriented practices, with scripts organized around class hierarchies and encapsulated behavior. On the other hand, the DOD version utilized the Entity Component System (ECS) approach, leveraging Unity's Burst compiler and Jobs package for efficient data processing and parallelization.

In terms of gameplay, the prototype successfully captured the essence of a bullet hell game. The player controlled a simple player who moves and shoots on demand to move across the enemy forces.

The prototype does not implement all the necessary or usual sections of bullet hell games but was designed with those in mind so it could easily be expanded and built upon without fully understanding the project's complexity (**Figure 44**).

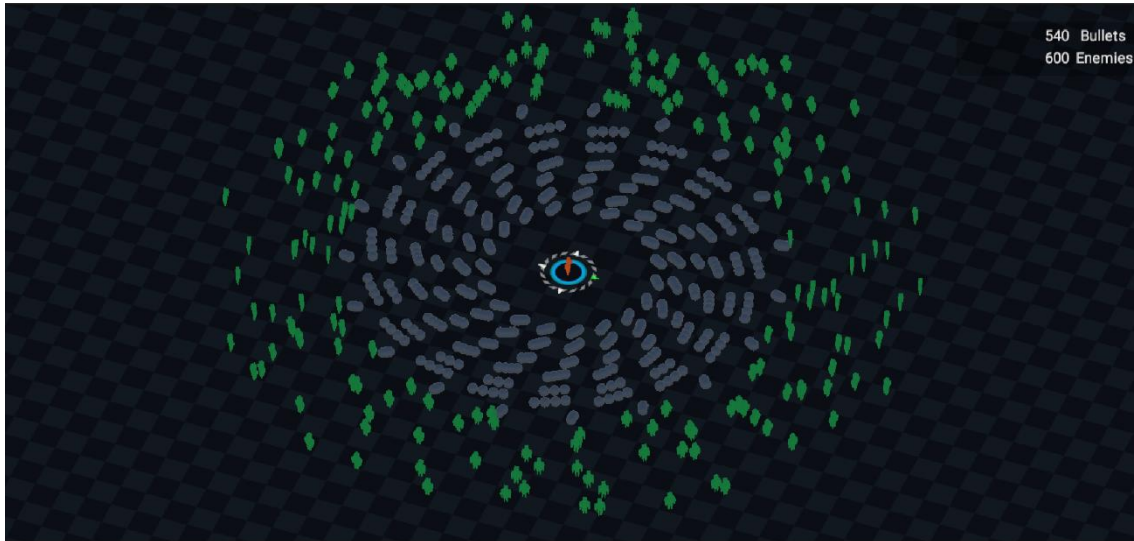


Figure 44: Showcasing of some bullet patterns heading for the enemies

The OOD version has more graphical features like a crosshair for aiming and some positional help, and its bullets have particle systems instead of just plain objects. These could not be implemented in the DOD version due to the project expanding multiple ECS versions, which changed the structure and API, creating a need to rebuild the scripts until the full release on June 2023, when the final prototype was finished. While the package is production-ready, animations, particles, and shapes like the crosshair are not yet supported, making the DOD project look plain and simple (**Figure 45**).



Figure 45: Comparison of players. OOD on the left, DOD on the right

Both projects were tested by creating bullets, spawning enemies, and checking the collisions. This was done on a 1920x1080@60Hz monitor, a GPU Nvidia GeForce RTX 2060, 6GB of VRAM, and 15GB of RAM with a CPU Intel® Core™ i7-10750H CPU @ 2.60Hz with 12 cores, and the project was compiled not to have editor performance interfere with the testing. An example test can be seen in **Figure 46** and **Figure 47** for OOD and DOD, respectively.

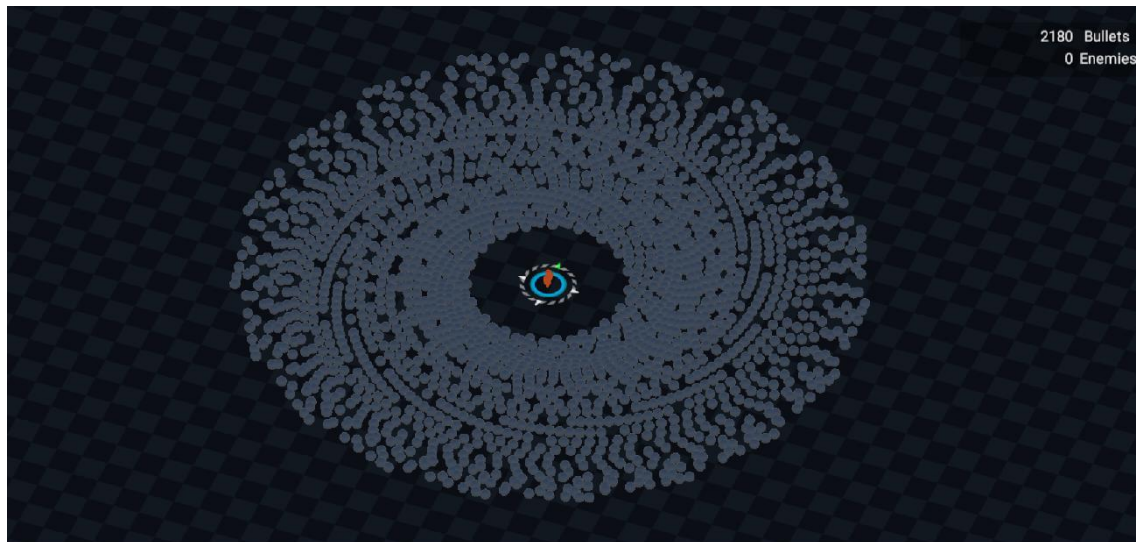


Figure 46: OOD runtime performance

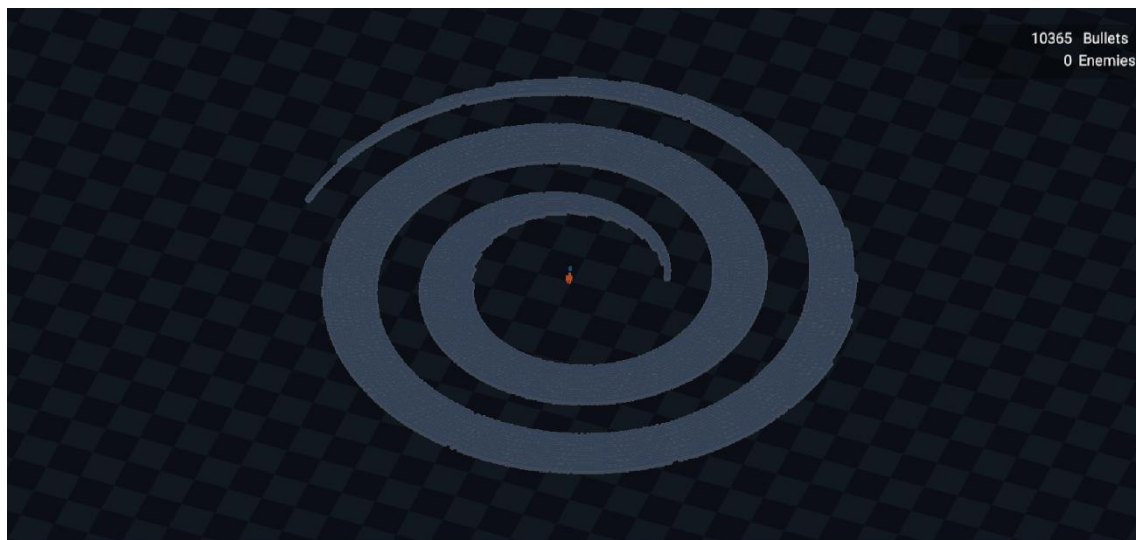


Figure 47: DOD runtime performance

In the subsequent sections of this research report, we will delve into the specific results and draw conclusions based on the performance analysis and feedback gathered during the prototype development.

More project images can be found in **Annex E**; the project files can be linked in **Appendix B**.

7.2. Comparative analysis

Four tests have been performed over two builds, one in each paradigm. These tests have been a bullet stress test, an enemy stress test, and a memory usage for each. All tests have been recorded with only the builds open, and the Unity profiler to record the data through some clutter on the RAM and CPU could not be avoided due to the lack of a proper benchmarking test machine.

7.2.1. Bullet stress test

The first test was done by calculating the mean Frames per Second (FPS) along 100 frames running with a given number of bullets. The complete datasets can be seen in **Appendix A**.

FPS is erratic, and calculating the mean was not easy, leading to only one result related to FPS (**Figure 48**). Since FPS is a metric calculated as $1000/\text{milliseconds}$ (ms), with ms being the time to execute the scripts, the tests moved to be ms dependent as the Unity profiler also provided it.

Even when stating that the method of comparison will change, we can see a clear pattern in the graph which is enhanced when seeing the ms comparisons in **Figure 49**. To put the graphs into perspective, $\sim 16ms$ is $60fps$, an acceptable rate for most video games, and $\sim 30ms$ is $30fps$, which is the test's goal. There will be two kinds of tests, managed and real, to distinguish where the complexity comes from. Managed only considers the ms resultant from managed code; the code I have created for the project, while real, will consider all ms from the rendering, lighting, and everything happening in the scene.

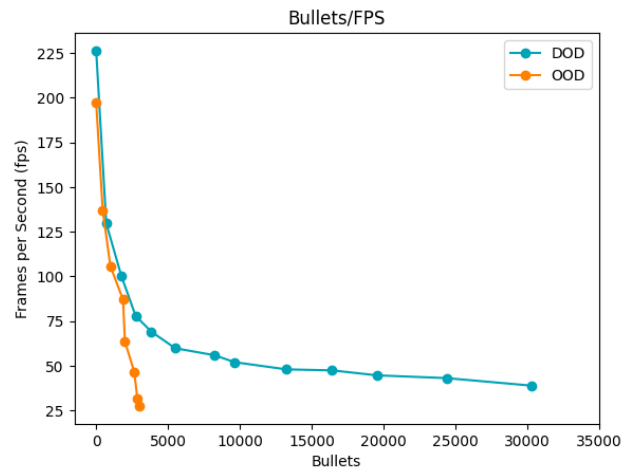


Figure 48: Frames per second as the number of bullets increases

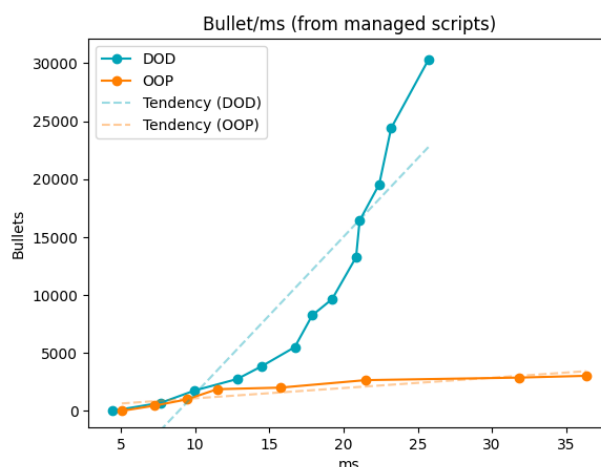


Figure 49: ms as the number of bullets increases for managed scripts

We can see a trend in both implementations as the number of bullets increases. While the OOP implementation does not seem to handle the new bullets well, and the code gets slower, the number of bullets barely increases. In contrast, the DOD did not reach the goal of *30ms* when the build reached the technical limit, as shown in **Figure 50**.

We find something similar in the real ms value: OOP looks like a logarithmic function, and DOD is more linear. When looking at the profiler, there were rendering spikes as more bullets were created, and, in the DOD case, it was a limiting factor as follow-up results comparing the enemies will delve into.

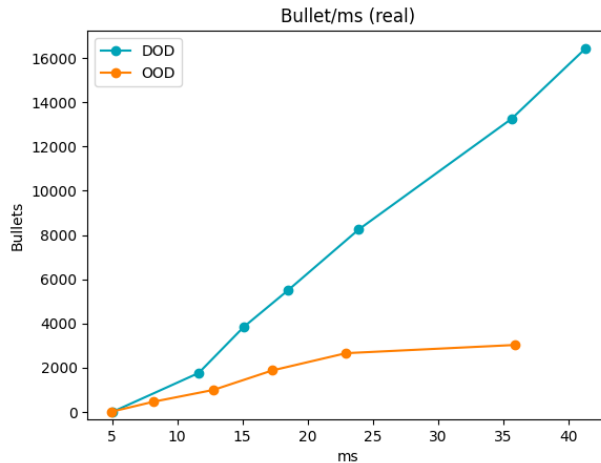


Figure 50: ms as the number of bullets increases for the whole frame

As someone with some knowledge in this area, I tried to make it as good as possible, but some concepts still needed improvement. I decided to share the prototype with the ECS community and asked for their feedback. One of the developers noticed that one of the DOD jobs needed to be Burst Compiled. It was a simple change that only required switching to a different compiler. However, this change had a massive impact on the performance.

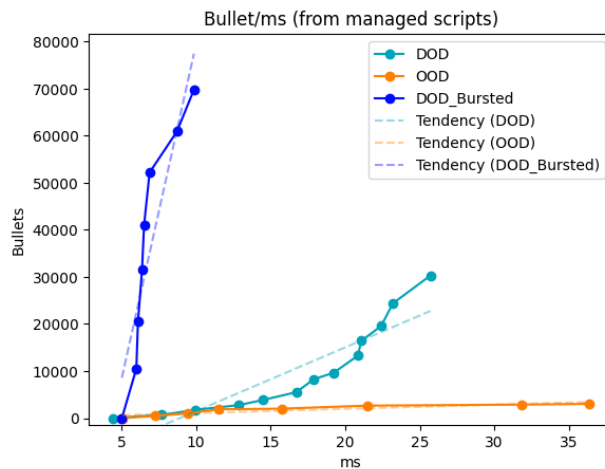


Figure 51: ms as the number of bullets increases for the managed scripts with new version

Figure 51 shows the new build, DOD_Bursted, which is the DOD version with the changes made to the mentioned job. The performance skyrocketed, and the test was only stopped because, like previously, other parts of the frame dragged the performance down. This can be fully seen in **Figure 52**, which had all three versions reach the target of *30ms* (*30fps*). When they reached the mark, OOD had ~ 2900 , DOD had ~ 13000 , and the new version had ~ 70000 bullets gaining a substantial quantity of bullets comparing versions.

Then came the memory tests over the actual build performance shown in **Figure 52**. They are taken from Unity's profiler, which has a memory section specific to OOD and DOD.

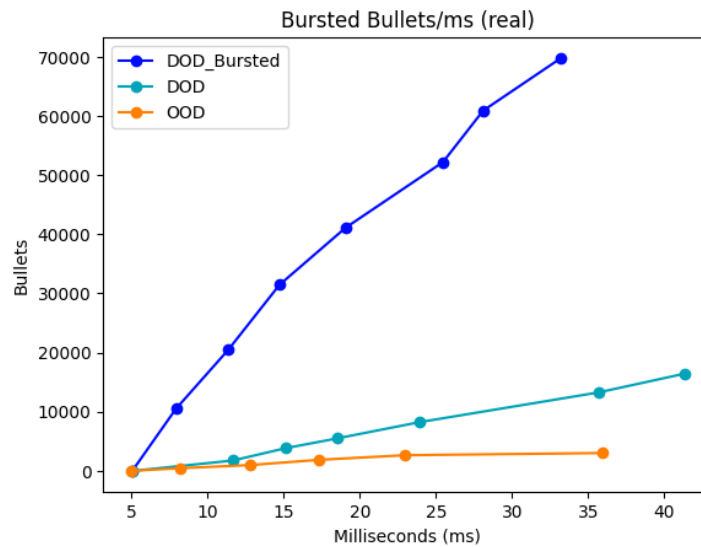


Figure 52: Final results for bullets and in-game times

Figure 53 shows that DOD has a linear data allocation as more memory is reserved to handle the increasing amount of bullet chunks since, as seen in **Figure 41**, they can only hold 20 each, so creating more bullets should and does have a linear increase. Even when the memory allocated is equal, the number of bullets managed is over 20 times the amount from the OOD version.

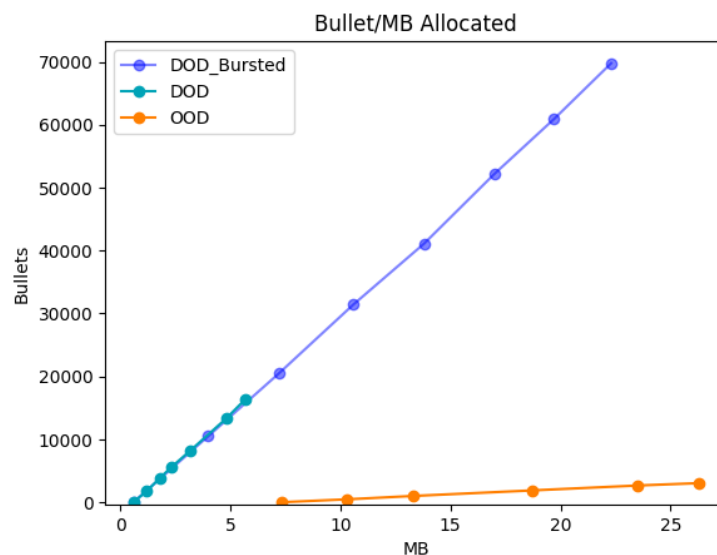


Figure 53: MB of data allocated as the number of bullets increases

These tests were repeated with the enemies as their behaviors are less complex in OOD and DOD implementations. In this case, the DOD_Bursted build did not differ from the original DOD since the changes were done on one of the bullet's jobs, so it has been removed for the following graphs.

7.2.2. Enemy stress test

Due to the data disparity between the two approaches, two bullet axis have been used, one for each implementation.

Figure 54 shows how DOD reached the limit once again on the managed scripts before it could reach the 30ms cost that was aimed to, while the OOD approach reached ~15000 enemies. This is more than 100 times the enemy amount when performance matched.

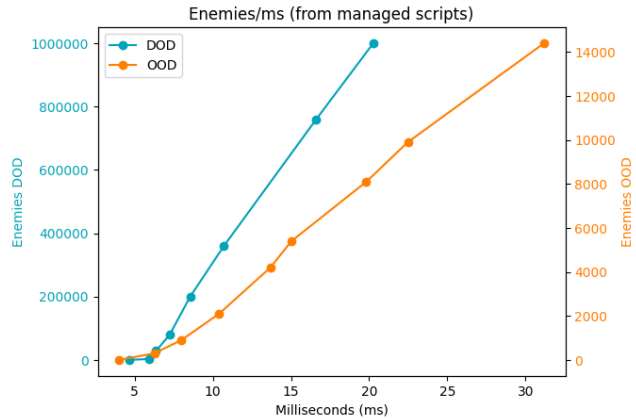


Figure 54: ms change over managed scripts as enemies increase

Outside the managed scripts and in the actual game environment, things did not change much but became closer.

Both paradigms reached the 30 ms goal but with wildly different results this time. DOD had ~30 times the amount of bullets when it reached the 30ms mark (**Figure 55**). Both enemy logics were the same for both paradigms but still had different results regarding quantity.

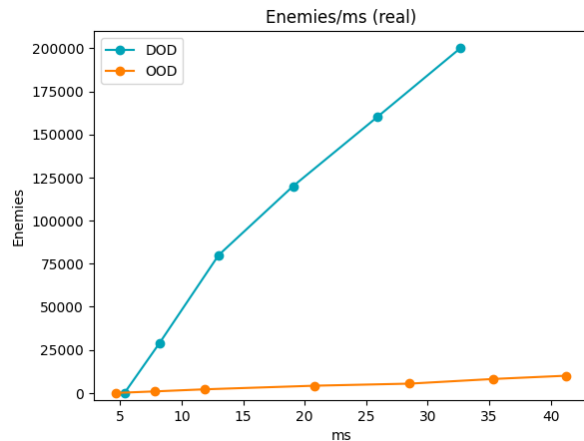


Figure 55: ms change over the full frame as enemies increase

Regarding the memory checks for the enemies, **Figure 56** shows a similar result to **Figure 53**, with DOD being linear as bullets were created and OOD needing much extra memory to handle a fraction of what DOD was handling at that time. Once again, the DOD approach stays coherent with its bases as data is created and managed according to the needs trying to minimize wasted data at all costs.

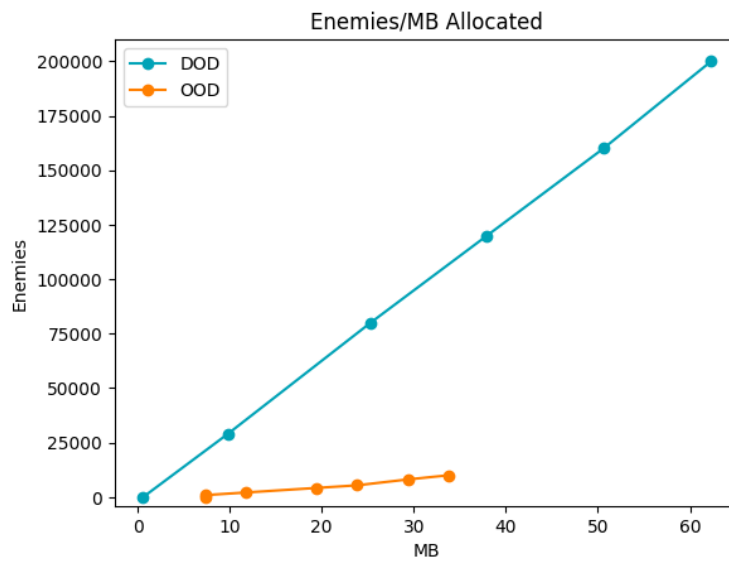


Figure 56: MB of data allocated as the number of enemies increases

Regarding all tests and their collection, **Annex F** offers screenshots of Unity's profiler and how data was acquired.

8. Validation Threats

During this research project, several potential validation threats should be acknowledged. These threats may have introduced biases or limitations impacting the findings' validity and generalizability. Recognizing and addressing these threats to minimize them in future works is essential.

- **Prototype Representativeness:** The prototype game developed for this project may only partially represent the complexities and intricacies of real-world video game development. The chosen use cases or game mechanics may not capture the diverse scenarios encountered in actual game projects. Further research should be done to create and accurately represent the real-world scenario.
- **Prototype Bias:** The choice of prototypes developed for the comparison may introduce a bias towards a particular paradigm. If the goal prototype favored one of the two paradigms more, it would create a disparity in the data. Further research on different video game types is needed to ensure a fair comparison.
- **Time/Length Constraints:** The project's time and length constraints may limit the depth and breadth of the analysis. The research may not have sufficient time to explore all possible aspects and variations of OOD and DOD implementation in video game development, and all insights may not have been shown due to the reporting limit. Extensive research without the constraints or with larger ones would give more leeway to tackle a bigger surrounding both paradigms.
- **Technical Limitations:** Due to the technology implemented being so new, it is not easy to assess the quality of the implementation. This needs to be tackled by making another comparison when the technology is fully stable and has all the essential components necessary to develop a prototype that can match the original implementation.
- **Scale and Complexity of Prototypes:** The scale and complexity of the prototypes developed for the project may impact the generalizability of the findings. Due to the relatively small and straightforward prototypes, the challenges and benefits associated with OOD and DOD may still need to be fully realized. Conversely, if the prototypes were overly complex and ambitious, it may introduce additional confounding factors that make it difficult to isolate the effects of the design paradigms. Mitigating this threat involves balancing prototype scale and complexity that aligns with real-world game development scenarios.

- **Expertise and Skill Level Variability:** The expertise and skill level acquired when implementing OOD and DOD can introduce variability and potential biases in the outcomes. Due to needing more knowledge about DOD to produce a matching prototype, the result may have required to be more balanced due to getting the scoop from more recent and specific sources than OOD. Before implementing the prototype without expert help, an extensive course should be done to ensure a matching knowledge ceiling.
- **Generalizability of Findings to Different Project Types:** The findings and conclusions drawn from this research project may only be fully generalizable to some types of video game development projects. The specific characteristics, requirements, and constraints of the chosen prototypes may limit the applicability of the findings to other game genres, platforms, or scales.

While most validation threats have been tackled through the process, some may still prevail and need future works to tackle and specify them correctly, leaving the door open for finer tuning and creating more knowledge on this subject.

9. Conclusions

Throughout this research project, we comprehensively analyzed and compared Object-Oriented Design (OOD) and Data-Oriented Design (DOD) in Unity game development. Our objective was to explore the strengths and weaknesses of these design paradigms and provide insights into their applicability and utilities in the gaming industry and the university one.

While OOD did not perform as well as DOD on any of the tests, it offers a friendly and intuitive approach, making it suitable for projects with moderate complexity and smaller team sizes where optimization may not be the key focus or a necessity or where the learning costs would be too high to be considered. On the other hand, DOD shines in situations that require parallel processing, optimizations, and scalable designs. It proves exceptionally advantageous for games with massive amounts of entities, intricate gameplay mechanics, and demanding performance requirements. This is mixed with the surrounding tools Unity provides, like the Job System or the Burst Compiler, that genuinely takes DOD to another level in performance.

Utilizing OOD and DOD in game development presents its own set of challenges. OOD's emphasis on abstraction and encapsulation can introduce additional layers of indirection and overhead, potentially impacting performance in specific scenarios. It requires careful consideration of class hierarchies and interdependencies while being flexible when the projects are small. While providing performance benefits, DOD may pose a learning curve for developers accustomed to traditional OOD practices. It demands a more direct and manual approach to code organization. It may require a shift in mindset that can require long learning times, hands-on work in the code directly, which may deter many people from trying, and specific tools not currently expected in the industry, as OOD has been the prevalent paradigm for universities and the industry.

Successfully transitioning from OOD to DOD requires critical concepts to be grasped and implemented effectively. Understanding the Entity Component System (ECS) architecture, utilizing the Jobs system for efficient parallel processing, and optimizing data layout are essential. Embracing the data-driven mindset and embracing the principles of cache coherency, data-streamlining, and low-level design is crucial to getting the desired optimization out of DOD.

While the transition did not oppose challenges outside the learning curve of the paradigm, this is mainly thanks to Unity's easy-to-use API and the abstraction from many inside actions that are

taken care of by the API directly. This eases the transformation of individual scripts but still requires work to optimize data layouts or dependencies, among other things.

Migrating an existing project from OOD to DOD incurs certain costs. It involves rewriting and restructuring code to align with the DOD principles. The complexity of the project, its size, and the familiarity of the development team with DOD concepts will profoundly influence the magnitude of the migration effort.

In conclusion, OOD and DOD offer unique advantages and trade-offs in game development. OOD provides a friendly and flexible approach suitable for beginner to complex projects that may be manageable. At the same time, DOD excels in optimizing data processing and performance for demanding game environments. The choice between the two paradigms depends on the specific needs, constraints of the project, and the prior knowledge of the people involved. Understanding the specific use cases, challenges, and key concepts involved in transitioning from OOD to DOD is essential for developers to make informed decisions and reduce the costs of transitioning from one paradigm to another.

9.1. Future Work

While this research project has shed light on comparing OOD and DOD in Unity game development, several avenues exist for future exploration and research. These key areas of future work aim to deepen our understanding and provide further insights into the application of OOD and DOD in game development, student academia, and industry standards.

Migration on a Large-Scale Project: This research focused on migrating from OOD to DOD for a prototype project. However, future research should explore the challenges and costs of migrating large-scale projects from OOD to DOD. Analyzing the impact on development time, performance optimization, and team coordination over DOD challenges and requirements would provide valuable insights for teams considering adopting DOD on existing projects.

Cost Analysis of DOD from Project Inception: While this research examined the cost implications of migrating an existing OOD project to DOD, conducting a cost analysis of developing a project using DOD from its creation would be valuable to compare if setting a base of work could lead to time gained in the creation time. This would involve creating a video game with DOD from the early stages of game development and comparing it to existing projects developed with OOD.

Performance Comparison in Different Game Genres: This research project focused on implementing a bullet hell prototype as a case study. Future work could explore the performance comparison between OOD and DOD in different game genres which may not seek such performant solutions, such as endless runners, tower defense, or puzzle-like games, and some that may require those performant solutions, such as open-world environments, multiplayer experiences, or physics-intensive simulations. Understanding how game genres and mechanics interact with OOD and DOD paradigms would provide developers with genre-specific insights.

Developer Learning Curve and Training: As DOD is still a relatively new concept for many game developers, future research could focus on understanding the learning curve associated with adopting and learning DOD. Exploring effective training methodologies, educational resources, and best practices for transitioning from OOD to DOD would facilitate a smoother adoption process and help developers overcome the initial challenges.

Impact of Student Capabilities on DOD Adoption: Another area of future work is studying the capabilities and aptitude of students or novice developers in understanding and implementing DOD concepts. Investigating how students with varying programming knowledge and experience grasp DOD principles would provide valuable insights into the accessibility and suitability of DOD as an educational approach in game development curricula. Additionally, exploring the effectiveness of different instructional methods and techniques for teaching DOD to students would enhance the educational resources available in this domain.

Evaluation of DOD Tools and Frameworks: As DOD gains traction in game development, the availability and usability of tools and frameworks that support DOD implementation should become increasingly important. Future research could assess the effectiveness and efficiency of existing DOD-oriented tools and frameworks outside Unity's, examining their impact on development productivity, code quality, and performance optimization.

9.2. Personal Learnings

This project has been the goal I started being curious about two years ago when I first asked my teachers for Data Oriented Design a semester after learning about Object Oriented Design. Since then, I have been curious about it, and it has pushed me to try languages like Rust which are more data aligned than C++.

Then I learned about Unity's focus on DOD, and since then, I have been fascinated by the possibilities and the uses the Unity team has showcased. ECS has been a part of my life since last year, 2022, and especially 2023, where I have interacted with the community, talked and learned from them, and soaked in as much knowledge as possible. I feel this is the first step into a more significant thing that I will grow accustomed to and genuinely love, as I cannot wait for summertime and do my projects regarding the paradigm used to squeeze the performance out of every game I make from now on.

I missed some guidance at first as no one had ideas or knowledge about ECS or DOD. It is vital to have the tools that DOTS provides in video game design, even with engines like Unreal trying to mask things and do them themselves under the hood. While it comes from my nature and likes, I think a minimum of knowledge would enhance every game and that the tools that have helped me the most have been taught mouth to mouth, in talks, and in some specific sources from the ECS developers like Andrew Parsons' tweet about learning resources which, while being published after the learning portion, it gave lots of insides on how things worked [115].

This project had consumed much of my life and provided immeasurable knowledge and pleasure, like the giggles when the builds were done, and the test were running. Seeing such disparities in performance, as the ECS team said would happen, felt like a pat on the back for understanding the concepts of an entirely different paradigm. It exceeded the initial goals as I became curious about cache allocations and how different definitions affected the performance, bringing a deeper understanding of Unity and coding.

I am grateful for choosing this research, and I am genuinely thrilled to see how my coding habits will change in the future.

10. References

- [1] R. Chikhani, "The History Of Gaming: An Evolving Community," TechCrunch+, 31 October 2015. [Online]. Available: <https://techcrunch.com/2015/10/31/the-history-of-gaming-an-evolving-community>. [Accessed 19 May 2023].
- [2] A. Marchand and T. Hennig-Thurau, "Value Creation in the Video Game Industry: Industry Economics, Consumer Benefits, and Research Opportunities," August 2013. [Online]. Available: https://www.researchgate.net/publication/255995598_Value_Creation_in_the_Video_Game_Industry_Industry_Economics_Consumer_Benefits_and_Research_Opportunities. [Accessed 19 May 2023].
- [3] A. McAloon, "Breaking down nearly 50 years of video game revenue," Game Developer, 30 January 2019. [Online]. Available: <https://www.gamedeveloper.com/console/breaking-down-nearly-50-years-of-video-game-revenue>. [Accessed 19 May 2023].
- [4] G. Koulaxidis and S. Xinogalos, "Improving Mobile Game Performance with Basic Optimization Techniques in Unity," March 2022. [Online]. Available: https://www.researchgate.net/publication/359597028_Improving_Mobile_Game_Performance_with_Basic_Optimization_Techniques_in_Unity. [Accessed 19 May 2023].
- [5] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome y D. Poshyvanyk, «Automatically Discovering, Reporting and Reproducing Android Application Crashes,» June 3017. [En línea]. Available: <https://arxiv.org/abs/1706.01130>. [Último acceso: 19 May 2023].
- [6] History.com Editors, "Video Game History," History, 17 October 2022. [Online]. Available: <https://www.history.com/topics/inventions/history-of-video-games>. [Accessed 19 May 2023].
- [7] K. Stuart, "The eight best advances in gaming during the last decade," The Guardian, 6 October 2017. [Online]. Available: <https://www.theguardian.com/technology/2017/oct/06/gaming-advances-decade-guardian-games-editor-keith-stuart>. [Accessed 19 May 2023].
- [8] K. Beck, «How video game development has changed over the last decade,» Mashable, 20 October 2019. [En línea]. Available: <https://mashable.com/article/video-game-development-over-the-decade>. [Último acceso: 19 May 2023].
- [9] M. Toftedahl, "Which are the most commonly used Game Engines?," 30 September 2019. [Online]. Available: <https://www.gamedeveloper.com/production/which-are-the-most-commonly-used-game-engines->. [Accessed 19 May 2023].
- [10] Unreal Engine, "Nanite Virtualized Geometry," 2022. [Online]. Available: <https://docs.unrealengine.com/5.0/en-US/nanite-virtualized-geometry-in-unreal-engine/>. [Accessed 19 May 2023].
- [11] K. Henney, "Giving Code a Good Name," 26 July 2017. [Online]. Available: <https://www.slideshare.net/Kevlin/giving-code-a-good-name>. [Accessed 19 May 2023].
- [12] R. Nystrom, Game Programming Patterns, Genever Benning, 2014.
- [13] R. Nystrom, "Design Patterns Revisited," 2021. [Online]. Available: <http://gameprogrammingpatterns.com/design-patterns-revisited.html>. [Accessed 19 May 2023].

-
- [14] R. Fabian, *Data-oriented design: software engineering for limited resources and short schedules*, 2018.
- [15] N. Llopis, "Data Alignment, Part 1," 06 March 2009. [Online]. Available: <https://www.gamedeveloper.com/programming/data-alignment-part-1>. [Accessed 19 May 2023].
- [16] N. Llopis, "Data Alignment, Part 2: Objects on The Heap and The Stack," 31 March 2009. [Online]. Available: <https://www.gamedeveloper.com/programming/data-alignment-part-2-objects-on-the-heap-and-the-stack>. [Accessed 19 May 2023].
- [17] N. Llopis, "Data-Oriented Design - Now And In The Future," 13 April 2011. [Online]. Available: <https://www.gamedeveloper.com/disciplines/sponsored-feature-data-oriented-design---now-and-in-the-future>. [Accessed 19 May 2023].
- [18] S. Meyers, "code::dive conference 2014 - Scott Meyers: Cpu Caches and Why You Care," code::dive, 2014. [Online]. Available: <https://youtu.be/WDIkqP4JbkE>. [Accessed 19 May 2023].
- [19] R. Joshi, «Data-Oriented Architecture: A Loosely-Coupled Real-Time SOA,» August 2007. [En línea]. Available: https://community.rti.com/sites/default/files/archive/Data-Oriented_Architecture.pdf. [Último acceso: 19 May 2023].
- [20] A. S. Tanenbaum and T. Austin, *Structured Computer Organization*, Pearson, 2013.
- [21] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, Pearson, 2015.
- [22] W. Faryabi, "Data-oriented Design approach for processor intensive games," 2 September 2018. [Online]. Available: <http://hdl.handle.net/11250/2575669>. [Accessed 19 May 2023].
- [23] Unity, "Entity Component System," Unity Docs, 19 October 2020. [Online]. Available: <https://docs.unity3d.com/Packages/com.unity.entities@0.11/manual/index.html>. [Accessed 19 May 2023].
- [24] Unity, "Harnessing the power of ECS for Unity | Unity at GDC 2023," Unity, 25 April 2023. [Online]. Available: <https://youtu.be/WSrvUynsd34?t=341>. [Accessed 19 May 2023].
- [25] Unity, "How to create, launch, and manage multiplayer games with Unity | Unity at GDC 2023," Unity, 11 April 2023. [Online]. Available: <https://youtu.be/h6fbgd-rzbE>. [Accessed 19 May 2023].
- [26] Unity, "Unity at devcom with IXION || Unity," Unity, August 2022. [Online]. Available: <https://www.youtube.com/watch?v=SjNE8BplBEk>. [Accessed 19 May 2023].
- [27] Unity, "JellyCar Worlds | Creator Spotlight | Made With Unity," Unity, December 2022. [Online]. Available: <https://www.twitch.tv/videos/1668085041>. [Accessed 19 May 2023].
- [28] Unity, "Creator Spotlight: V Rising | Made With Unity," Unity, June 2022. [Online]. Available: <https://www.twitch.tv/videos/1517796421>. [Accessed 19 May 2023].
- [29] Unity, "The path to leveraging DOTS in production | Unity at GDC 2022," Unity, 2 April 2022. [Online]. Available: <https://youtu.be/KJOhIhOv2EI>. [Accessed 19 May 2023].
- [30] L. Gibert, "DOTS development status and next milestones - June 2023," Unity, 01 June 2023. [Online]. Available: <https://forum.unity.com/threads/dots-development-status-and-next-milestones-june-2023.1443250/>. [Accessed 01 June 2023].
- [31] Unity, "ECS for Small Things," Unity, 23 March 2018. [Online]. Available: <https://youtu.be/EWVU6cFdmr0?t=1385>. [Accessed 19 May 2023].
-

-
- [32] P. Hu and K. Zhu, "Strategy research on the performance optimization of 3D mobile game development based on Unity," *Journal of Chemical and Pharmaceutical Research*, vol. 6, no. 3, pp. 785-791, 2014.
- [33] Unity, L. Gilbert and J. Valenzuela, "Unity LTS 2022: Introduction to DOTS," Unity, 22 June 2023. [Online]. Available: <https://www.youtube.com/live/oUQapNQgpRI?feature=share&t=508>. [Accessed 23 June 2023].
- [34] Unity, L. Gilbert and T. Johansson, "Unity LTS 2022: Deliver a seamless multiplayer experience," Unity, 22 June 2023. [Online]. Available: <https://www.youtube.com/live/oUQapNQgpRI?feature=share&t=9819>. [Accessed 23 June 2023].
- [35] N. Collman, "MAKU: A Code Generator for Bullet Hell Games," 2014.
- [36] Touhou Wiki, "The Shattered Sky/Spell Cards/Last Word," Touhou Wiki, 10 January 2022. [Online]. Available: https://en.touhouwiki.net/wiki/The_Shattered_Sky/Spell_Cards/Last_Word. [Accessed 23 June 2023].
- [37] S. Johnson, "Dodonpachi," *Hardcore Gaming 101*, 20 February 2015. [Online]. Available: <http://www.hardcoregaming101.net/dodonpachi/>. [Accessed 23 June 2023].
- [38] Touhou Wiki, "Touhou Project," Touhou Wiki, 11 May 2023. [Online]. Available: https://en.touhouwiki.net/wiki/Touhou_Wiki. [Accessed 23 June 2023].
- [39] A. C. Kay, "The Early History Of Smalltalk," *ACM SIGPLAN Notices*, vol. 28, no. 3, pp. 69-95, 01 March 1993.
- [40] A. Kay, "The Power Of The Context," Viewpoints Research Institute, California, 2004.
- [41] H. Hsu, "48 YEARS OF SMALLTALK HISTORY AT CHM," Computer History Museum, 2022. [Online]. Available: <https://computerhistory.org/blog/introducing-the-smalltalk-zoo-48-years-of-smalltalk-history-at-chm/>. [Accessed 31 May 2023].
- [42] ORACLE, "Chapter 4. Types, Values, and Variables," ORACLE, 03 March 2023. [Online]. Available: <https://docs.oracle.com/javase/specs/jls/se20/html/jls-4.html#jls-4.3.1>. [Accessed 31 May 2023].
- [43] ORACLE, "Autoboxing and Unboxing," ORACLE, 2022. [Online]. Available: <https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html>. [Accessed 31 May 2023].
- [44] Microsoft, "https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/oop," Microsoft, 01 March 2023. [Online]. Available: <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/oop>. [Accessed 31 May 2023].
- [45] B. Stroustrup, *A Tour of C++ (Second edition)*, Addison-Wesley, 2018.
- [46] B. Stroustrup, "The C++ Programming Language," 19 October 2021. [Online]. Available: <https://stroustrup.com/C++.html#guidelines>. [Accessed 31 May 2023].
- [47] G. v. Rossum, *Python Programming Language*, USENIX, 2007.
- [48] R. C. Martin, *Designing object-oriented C++ applications: using the Booch method*, 1995.
- [49] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, United States: Prentice Hall PTR, 2003.
- [50] C. Larman, *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process*, Prentice Hall Professional, 2002.
-

-
- [51] cplusplus.com, "C++ Language Polymorphism," cplusplus, 2022. [Online]. Available: <https://legacy.cplusplus.com/doc/tutorial/polymorphism/>. [Accessed 19 May 2023].
- [52] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Pearson Education, 1994.
- [53] ANECA, *Libro Blanco: Titulo de Grado en Ingenieria Informatica*, ANECA, 2004.
- [54] N. Tatarchuk, "A Flexible, On-the-Fly Object Manager," in *Game Programming Gems 4*, Charles River Media, 2004, pp. 103-110.
- [55] W. B. Frakes and R. Anguswamy, "A Study of reusability, complexity, and reuse design principles," *International Symposium on Empirical Software Engineering and Measurement*, vol. 12, pp. 161-164, 2012.
- [56] T. Sweeney, "The next mainstream programming language: a game developer's perspective," *ACM SIGPLAN Notices*, vol. 41, no. 1, p. 269, 2006.
- [57] M. Leair and S. Pande, "Optimizing dynamic dispatches through type invariant region analysis," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2913, pp. 459-468, 2003.
- [58] T. Yiyu, A. S. Fong y Y. Xiaojian, «Architectural solution to object-oriented programming,» *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4697, pp. 387-398, 2007.
- [59] Unity, "Understand data-oriented design," Unity Learn, 02 March 2023. [Online]. Available: <https://learn.unity.com/tutorial/part-1-understand-data-oriented-design>. [Accessed 02 June 2023].
- [60] J. Boer, "Object-Oriented Programming and Design Techniques," in *Game Programming Gems 1*, Charles River Media, 2000, pp. 8-19.
- [61] N. H., D. Armstrong and K. Nelson, "Patterns of transition: The shift from traditional to object-oriented development," *Journal of Management Information Systems*, vol. 25, no. 4, pp. 271-298, 2008.
- [62] J. Blow, "Data-Oriented Demo: SOA, composition," 2015.
- [63] N. Llopis, "Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP)," 4 December 2009. [Online]. Available: <https://gamesfromwithin.com/data-oriented-design>. [Accessed 31 May 2023].
- [64] M. Acton, "Data-Oriented Design and C++," in *CppCon*, 2014.
- [65] Unity, "We're joining Unity to help democratize data-oriented programming," Unity, 8 November 2017. [Online]. Available: <https://blog.unity.com/community/were-joining-unity-to-help-democratize-data-oriented-programming>. [Accessed 31 May 2023].
- [66] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice Hall, 1988.
- [67] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley Professional, 2001.
- [68] D. Kusswurm, *Modern Parallel Programming with C++ and Assembly Language: X86 SIMD Development Using AVX, AVX2, and AVX-512*, Apress, 2022.
- [69] A. Fredriksson, "SIMD at Insomniac Games," in *Game Developers Conference*, 2015.
- [70] M. Scarpino, "Crunching Numbers with AVX and AVX2," Code Project, 2 April 2016. [Online]. Available: <https://www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX>. [Accessed 31 May 2023].
-

-
- [71] D.-o. d. s. e. f. l. r. a. s. schedules, *Data-oriented design: software engineering for limited resources and short schedules*, Manning Publications Co., 2022.
- [72] U. Drepper, "What Every Programmer Should Know About Memory," in *Drepper2007WhatEP*, 2007.
- [73] N. Ivanova and D. Charrieras, "Emergence in video game production: Video game engines as technical individuals," *Social Science Information*, vol. 55, no. 3, pp. 337-356, 2016.
- [74] T. A. Fontana, S. F. Almeida, R. Netto, V. S. Livramento, C. Guth, L. L. Pilla and J. L. A. Güntzel, "Exploiting cache locality to speedup register clustering," in *SBCCI '17*, Brazil, 2017.
- [75] S. Rabin, "The Magic of Data-Driven Design," in *Game Programming Gems 1*, Charles River Media, 2000, pp. 3-7.
- [76] T. Mironov, L. Motaylenko, D. Andreev, I. Antonov and M. Aristov, "Comparison of object-oriented programming and data-oriented design for implementing trading strategies backtester," *Vide. Tehnologija. Resursi - Environment, Technology, Resources*, vol. 2, pp. 124-130, 2021.
- [77] D. Wingqvist, F. Wickstrom and S. Memeti, "Evaluating the performance of object-oriented and data-oriented design with multi-threading in game development," *2022 IEEE Games, Entertainment, Media Conference, GEM 2022*, 2022.
- [78] K. Fedoseev, N. Askarbekuly, E. Uzbekova and M. Mazzara, "Application of Data-Oriented Design in Game Development," *Journal of Physics: Conference Series*, vol. 1694, no. 1, pp. 12-35, January 2020.
- [79] I. Seah and L. White, "Havok Physics for Unity is now supported for production," Unity Blog, 19 December 2022. [Online]. Available: <https://blog.unity.com/engine-platform/havok-physics-now-supported-for-production>. [Accessed 25 May 2023].
- [80] R. Barringer, "Data-oriented rigid body physics," 17 June 2017. [Online]. Available: <https://rasmusbarr.github.io/blog/dod-physics.html>. [Accessed 25 May 2023].
- [81] Unity, "Entity Component System - Unity Learn," Unity Learn, 02 May 2023. [Online]. Available: <https://learn.unity.com/tutorial/entity-component-system#>. [Accessed 25 May 2023].
- [82] RegularX, "Unreal Object Oriented Programming," 23 June 2007. [Online]. Available: <https://beyondunrealwiki.github.io/pages/unreal-object-oriented-prog.html>. [Accessed 25 May 2023].
- [83] Unity, "Unity's Data-Oriented Technology Stack (DOTS)," 2023. [Online]. Available: <https://unity.com/dots>. [Accessed 25 May 2023].
- [84] Unity, "Data design - Unity Learn," 03 April 2023. [Online]. Available: <https://connect-prd-cdn.unity.com/20210202/3b84b9c2-d8b1-465e-88b9-41dcc11e205b/Breakout%20Data%20Worksheet.pdf>. [Accessed 02 June 2023].
- [85] Unity, "Implementation fundamentals - Unity Learn," Unity, 02 March 2023. [Online]. Available: <https://learn.unity.com/tutorial/part-3-1-implementation-fundamentals>. [Accessed 02 June 2023].
- [86] J. D. Bayliss, "Computer," *Computer*, vol. 55, no. 5, pp. 31-38, 2022.
- [87] Unity Docs, "About Burst," 22 May 2023. [Online]. Available: <https://docs.unity3d.com/Packages/com.unity.burst@1.8/manual/index.html>. [Accessed 25 May 2023].
-

-
- [88] T. Jones, "Raising your game with Burst 1.7," Unity, 14 May 2022. [Online]. Available: <https://blog.unity.com/engine-platform/raising-your-game-with-burst-1-7>. [Accessed 02 June 2023].
- [89] P. Ananga and I. K. Biney, "COMPARING FACE-TO-FACE AND ONLINE TEACHING AND LEARNING IN HIGHER EDUCATION," *MIER Journal of Educational Studies, Trends & Practices*, vol. 7, no. 2, pp. 165-179, November 2017.
- [90] I. Tort-Ausina, J. Gómez-Tejedor, J. Molina-Mateo, J. Riera, J. Meseguer-Dueñas, R. Martín-Cabezuelo and A. Vidaurre, "Results Of a University Experience, Comparing Face-To-Face, Online and Hybrid Teaching in a Context of SARSCOV19," in *EDULEARN22 Conference*, Mallorca, 2022.
- [91] J. Petchamé, I. Iriondo, E. Villegas, D. Riu and D. Fonseca, "Comparing Face-to-Face, Emergency Remote Teaching and Smart Classroom: A Qualitative Exploratory Research Based on Students' Experience during the COVID-19 Pandemic," *Sustainability*, vol. 13, 2021.
- [92] Unity Docs, "Conversion Workflow," 6 July 2022. [Online]. Available: <https://docs.unity3d.com/Packages/com.unity.entities@0.50/manual/conversion.html>. [Accessed 25 May 2023].
- [93] Unity Docs, "Understand the ECS workflow," 25 May 2023. [Online]. Available: <https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/ecs-workflow-intro.html>. [Accessed 2023 May 2023].
- [94] James, "DanmakU - Bullet Hell Development Kit," 2015.
- [95] "Bagoum", "Danmokou Documentation," 2023.
- [96] V. R. Basili, G. Caldiera and H. D. Rombach, "The Goal Question Metric Approach," in *Basili1994TheGQ*, 1994.
- [97] G. Travassos and M. Barros, "Contributions of In Virtuo and In Silico Experiments for the Future of Empirical Studies in Software Engineering Contributions of In Virtuo and In Silico Experiments for the Future of Empirical Studies in Software Engineering," *The Future of Empirical Studies in Software Engineering: Proceedings of the ESEIW 2003 Workshop on Empirical Studies in Software Engineering, WSESE 2003*, vol. 2, p. 117, 2004.
- [98] J. Dunstan, "How to Write Faster Code Than 90% of Programmers," 8 May 2017. [Online]. Available: <https://www.jacksondunstan.com/articles/3860>. [Accessed 23 June 2023].
- [99] J. Nielson, "Nova Drift Dev Deep Dive: Dynamic Waves!," Nova Drift Blog, 18 October 2021. [Online]. Available: <https://blog.novadrift.io/nova-drift-dev-deep-dive-dynamic-waves/>. [Accessed 2023 06 23].
- [100] Odin, "Odin Inspector: Improve your workflow in Unity," Odin, 2023. [Online]. Available: <https://odininspector.com/>. [Accessed 23 June 2023].
- [101] Turbo Makes Games, "https://www.youtube.com/watch?v=Bz24Jp30nkM&t," Turbo Makes Games, 23 June 2023. [Online]. Available: <https://www.youtube.com/watch?v=Bz24Jp30nkM&t>. [Accessed 23 June 2023].
- [102] Unity, "Unity Entities package 1.0 - Entities and Components," Unity, 28 September 2022. [Online]. Available: <https://www.youtube.com/watch?v=jzCEzNoztzM>. [Accessed 25 June 2023].
- [103] Unity, "Use enableable components," Unity, 21 June 2023. [Online]. Available: <https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/components-enableable-use.html>. [Accessed 25 June 2023].
-

-
- [104] Unity, "Burst User Guide," Unity, 2020. [Online]. Available: <https://docs.unity3d.com/Packages/com.unity.burst@1.2/manual/index.html>. [Accessed 25 June 2023].
- [105] Unity, "Burst - C#/.NET type support," Unity, 12 June 2023. [Online]. Available: <https://docs.unity3d.com/Packages/com.unity.burst@1.8/manual/csharp-type-support.html>. [Accessed 25 June 2023].
- [106] Unity, "The Unity Job System," Unity, 28 September 2022. [Online]. Available: <https://www.youtube.com/watch?v=jdW66hA-Qu8>. [Accessed 25 June 2023].
- [107] Unity, "Physics project setup," Unity, 21 June 2023. [Online]. Available: <https://docs.unity3d.com/Packages/com.unity.physics@1.0/manual/getting-started-installation.html>. [Accessed 25 June 2023].
- [108] Unity, "Enum TransformUsageFlags," Unity, 21 June 2023. [Online]. Available: <https://docs.unity3d.com/Packages/com.unity.entities@1.0/api/Unity.Entities.TransformUsageFlags.html>. [Accessed 25 June 2023].
- [109] Unity, "Unity 2022 LTS is here! | Unity," Unity, 1 June 2023. [Online]. Available: <https://www.youtube.com/watch?v=5OgvVVQyur8&t>. [Accessed 25 June 2023].
- [110] L. Gibert, "DOTS: get in touch with the teams behind it!," Unity, 09 May 2023. [Online]. Available: <https://forum.unity.com/threads/dots-get-in-touch-with-the-teams-behind-it.1434796/>. [Accessed 25 June 2023].
- [111] PayScale, «Average Mid-Career Software Engineer / Developer / Programmer Salary in Spain,» PayScale, 2023. [En línea]. Available: https://www.payscale.com/research/ES/Job=Software_Engineer_%2F_Developer_%2F_Programmer/Salary/0749c27a/Mid-Career. [Último acceso: 25 June 2023].
- [112] Hello Astra, "Average Software Developer salary in 2023 in the EU," Hello Astra, 30 January 2023. [Online]. Available: <https://helloastra.com/blog/article/average-software-developer-salary-in-2023-in-the-eu>. [Accessed 25 June 2023].
- [113] E. Kononchuk, "Average Software Developer Salaries: Rates Comparison by Country," Qubit Labs, 09 May 2023. [Online]. Available: <https://qubit-labs.com/average-software-developer-salaries-salary-comparison-country/>. [Accessed 25 June 2023].
- [114] Y. S. a. C. European Commission and Directorate-General for Education, ECTS users' guide 2015, Publications Office, 2017.
- [115] A. Parsons, "My current top ten for all things DOTS & ECS," Twitter, 11 May 2023. [Online]. Available: <https://twitter.com/MrAndyPuppy/status/1656329064242884610?s=20>. [Accessed 25 June 2023].
- [116] A. Hamlett, "WakaTime," WakaTime, September 2022. [Online]. Available: <https://wakatime.com/>. [Accessed 23 June 2023].
- [117] WakaTime and D. Muñoz, "Test_ECS_1_08," WakaTime, 23 June 2023. [Online]. Available: <https://wakatime.com/@66a229f4-c9eb-4223-9123-d748b55e5ecb/projects/egbtvxfbl>. [Accessed 23 June 2023].
- [118] Wikipedia, "Touhou Project," 06 June 2023. [Online]. Available: https://en.wikipedia.org/wiki/Touhou_Project. [Accessed 23 June 2023].

11. Appendix

11.1. Appendix A

11.1.1. Datasets

The complete datasets can be found in [this link](#).

They are divided into *managed* and *real*. This separation is related to which ms is being evaluated. Managed means only the scripts created by the programmer are considered, seeing the growth the algorithms and mechanics pose in the project. *Real* considers everything that needs to happen, each frame showing the result, which the player sees.

All the data has been taken as a mean from 100 frames running with the amount specified.

An example of the data is Enemies_Managed:

OOD	ms	DOD	ms
0	4	0	4.63
300	6.26	3000	5.91
900	7.98	29000	6.37
2100	10.4	80000	7.24
4200	13.68	200000	8.55
5400	15.03	360000	10.71
8100	19.82	760000	16.62
9900	22.48	1000000	20.27
14400	31.2	-	-

OOD and DOD columns mean the number of, in this case, enemies spawned for that paradigm. The ms columns register the number of ms the, in this case, managed scripts took to run their calculations.

Apart from these, the time to code each script has been added in the Time_Per_Paradigm file showing each significant (>1 minute invested) script and its paradigm. This comes from an extension that tracks the coding time of each file called WakaTime [116]. In the free version, only the last two weeks of work could be seen, which was circumvented by doing it twice as the main project took around one month of development and then adding the data manually from the dashboards [117].

11.2. Appendix B

11.2.1. Project files and builds

Due to the project's size, the entire project cannot be directly shared. Instead, it is stored online in [this link](#), and the builds are stored in [this link](#).

There are two published builds, **OOD2** and **DOD3**.

OOD2 is the Object Oriented version of the prototype.

DOD3 is the Data-Oriented version of the prototype.

Both have a *README* file explaining how to use and test them.

12. Annexes

12.1. Annex A: Propuesta de proyecto

Nombre alumno: Daniel Muñoz Muñoz

Titulación: Grado en Diseño y Desarrollo de Videojuegos

Curso académico: 2022 - 2023

1. TÍTULO DEL PROYECTO

Comparing Object Oriented Programming (OOP) and Data Oriented Programming (DOP) for video games.

2. DESCRIPCIÓN Y JUSTIFICACIÓN DEL TEMA A TRATAR

El proyecto consiste en estudiar y comparar, en el contexto del motor comercial Unity, la aplicación al diseño y desarrollo de videojuegos de:

- Programación Orientada a Objetos
- Programación Orientada a Datos

Con tal objetivo, además, el proyecto incluye dos prototipos de un mismo videojuego perteneciente al género *bullet hell*, utilizados para llevar a cabo el estudio y comparación descritos.

3. OBJETIVOS DEL PROYECTO

Los objetivos del proyecto son:

- Estudio del arte sobre OOP en la actualidad.
- Estudio del arte sobre DOP enfocado en el paquete de Unity Data Oriented Technology Stack (DOTS) en la actualidad.
- Diseño y desarrollo de un *vertical slice* de un *bullet hell* mediante la aplicación de OOP.
- Realización de un *vertical slice* equivalente diseñado mediante DOP, utilizando DOTS.
- Comparación de ambos prototipos a nivel de consumo de recursos y rendimiento.
- Recopilación de datos obtenidos mediante la comparación y posterior análisis de resultados.

4. METODOLOGÍA

La metodología a usar se desarrollará en la primera reunión con el director del proyecto.

5. PLANIFICACIÓN DE TAREAS

Las tareas quedan predefinidas en los objetivos. Se fijarán tareas concretas para alcanzarlas durante el desarrollo del proyecto.

6. OBSERVACIONES ADICIONALES

El director del proyecto será Daniel Blasco Latorre.

12.2. Annex B: Meetings record

Here is the record of formal meetings between Daniel Blasco (research tutor) and me, Daniel Muñoz. We had talks around the university campus and emails to speed things up, only setting meetings for important milestones or planning.

1. **Date:** 10th of November, 2022. **Type:** Online. **Place:** Teams

We set the bases for the project, expectations, general deadlines, and the workflow to follow. We discussed the implications of working with a developing technology and to try and aim the research as close to a research paper as it was possible while following the guidelines.

2. **Date:** 17th of February, 2023. **Type:** In person. **Place:** USJ

We talked about progress so far, state of the art, and how the technology had changed over the months since the last meeting. We set guidelines for the color scheme to try and follow when doing the schemes, the diagrams needed to explain things adequately, the extra work needed that might not be directly in this research but could help, and some general guidelines for writing like subsections and overall structure.

3. **Date:** 11th of May, 2023. **Type:** Online. **Place:** Teams

With the prototype well started and the technology almost ready to be fully used, we talked about the implementation details like the bullet patterns and what they would need to make them as close to a bullet hell style of game as well as some possible figures to showcase those implementations.

4. **Date:** 19th of June, 2023. **Type:** Online. **Place:** Teams

With the prototype fully finished, we talked about some key components of the research, like data availability, research position in time, legibility of the sections, showing the results, and the few sections that still needed to be finished.

5. **Date:** 23rd of June 2023. **Type:** Online. **Place:** Teams

With the research document needing the last touches, we talked about the little things to add, like new documents, possible references that could support some sections, and the latest version of the technology, as the day before, there had been an important event at Unity that gave more information about the tools used in this research (ECS).

12.3. Annex C

12.3.1. Data worksheet

Tables showing the representation of data in Data Oriented Design structures.

Data					
Data	Type	Quantity	Read Frequency	Write Frequency	Why do you need this data?
Position	float2	1 Ball Many Blocks 1 Paddle	Ball: Every Frame Blocks: Every Frame for Physics Paddle: Every Frame	Ball: Every Frame Blocks: Once on init Paddle: Every Frame	We need to track where the ball is
Size	float2 (width and height)	1 Ball Many Blocks 1 Paddle	Ball: Every Frame Blocks: Every Frame Paddle: Every Frame	Once on init	We need a width and height for each object in order to calculate collisions
Speed	float	1 Ball 1 Paddle	Ball: Every Frame Paddle: Input Change	Once on init	Combined with Direction can give you the entity's velocity, separated components because Direction will update while Speed doesn't
Direction	float2	1 Ball 1 Paddle	Ball: Every Frame Paddle: Input Change	Ball: On Collision Paddle: Input Change	Combined with Speed can give you the entity's velocity, separated because Direction will update while Speed doesn't
Color	float4	Many Blocks	Rendering Only	Once on init	This could be a component to allow for different colored blocks
Score	int	1 Ball	UI Updates when Score Updates	On Ball/Block Collision	We need to keep track of the current score to display to the player
Board	float2 (width and height)	1 Board	Read by Ball and Paddle to stay inbounds of the board	Once on init	We need to know the size of the game board to constrain the paddle and ball to move within the boundaries
PaddleTag		1 Paddle	Read when updating the Paddle's movement	Once on init	We need a way to differentiate movement driven by player input

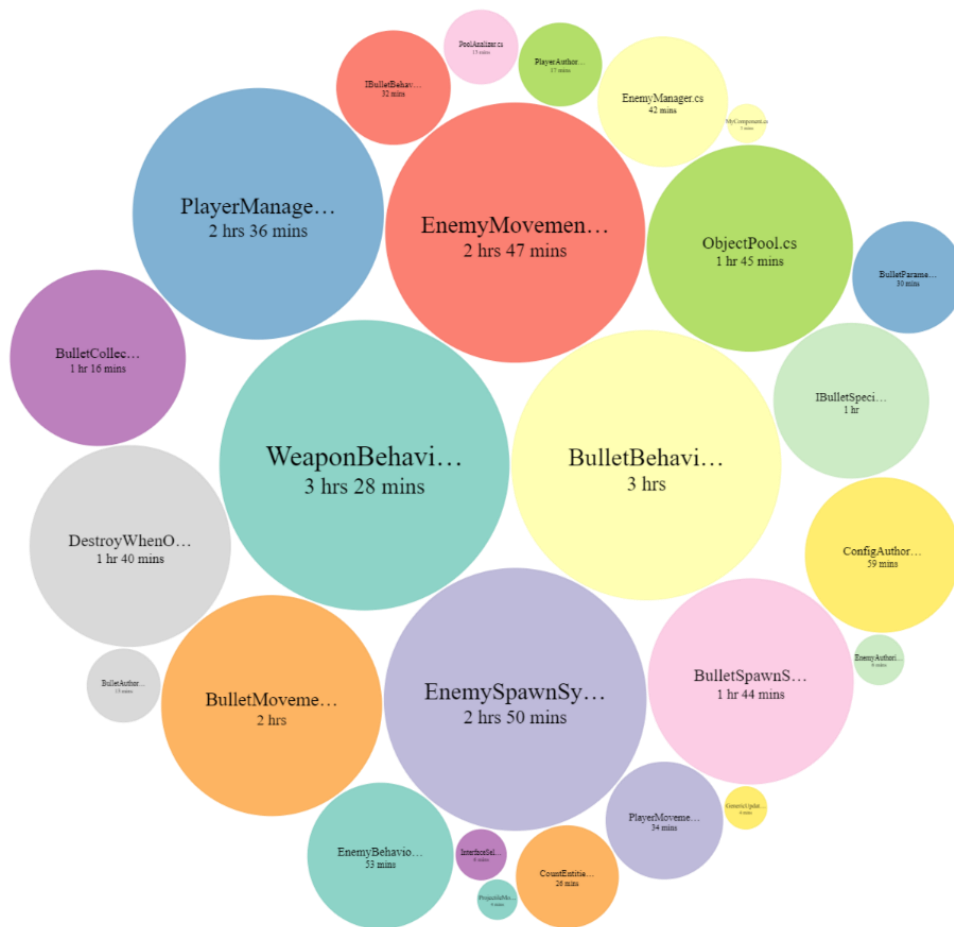
Data Transformations

Data Input	Output	System	When and how frequently does this occur?	What other data do you need?
Position Speed Direction Board Not PaddleTag	Position Direction (If at edge of board)	Ball Movement	Once every frame, Position always updated, Direction updated if hits edge of the board (Dies if hits the bottom)	
Size Position	Direction (Ball) Destroy Block (If hit)	Block Collision Score (If hit)	Once every frame	Ball Direction Score
Position Speed Direction Size PaddleTag	Direction (If input) Position Speed (0 when no input)	Paddle Movement	Once every frame	Input.Axis
Board	Position Size Color	Block Spawning	Once on init	Block Prefab Number of Rows
Position Direction Speed	Position Direction Speed Size	Ball Spawning	Once on init	Ball Prefab Any authoring data
Position Direction Speed	Position Direction Speed Size PaddleTag	Paddle Spawning	Once on init	Paddle Prefab Any authoring data
Position Size Color	Draw calls	Rendering	Once every frame	Some entities (e.g. Ball, Paddle) don't have a Color component. Default to white.

12.4. Annex D

12.4.1. Time description per script

A bubble map generated by WakaTime shows the time taken to develop each script of the researches prototype:

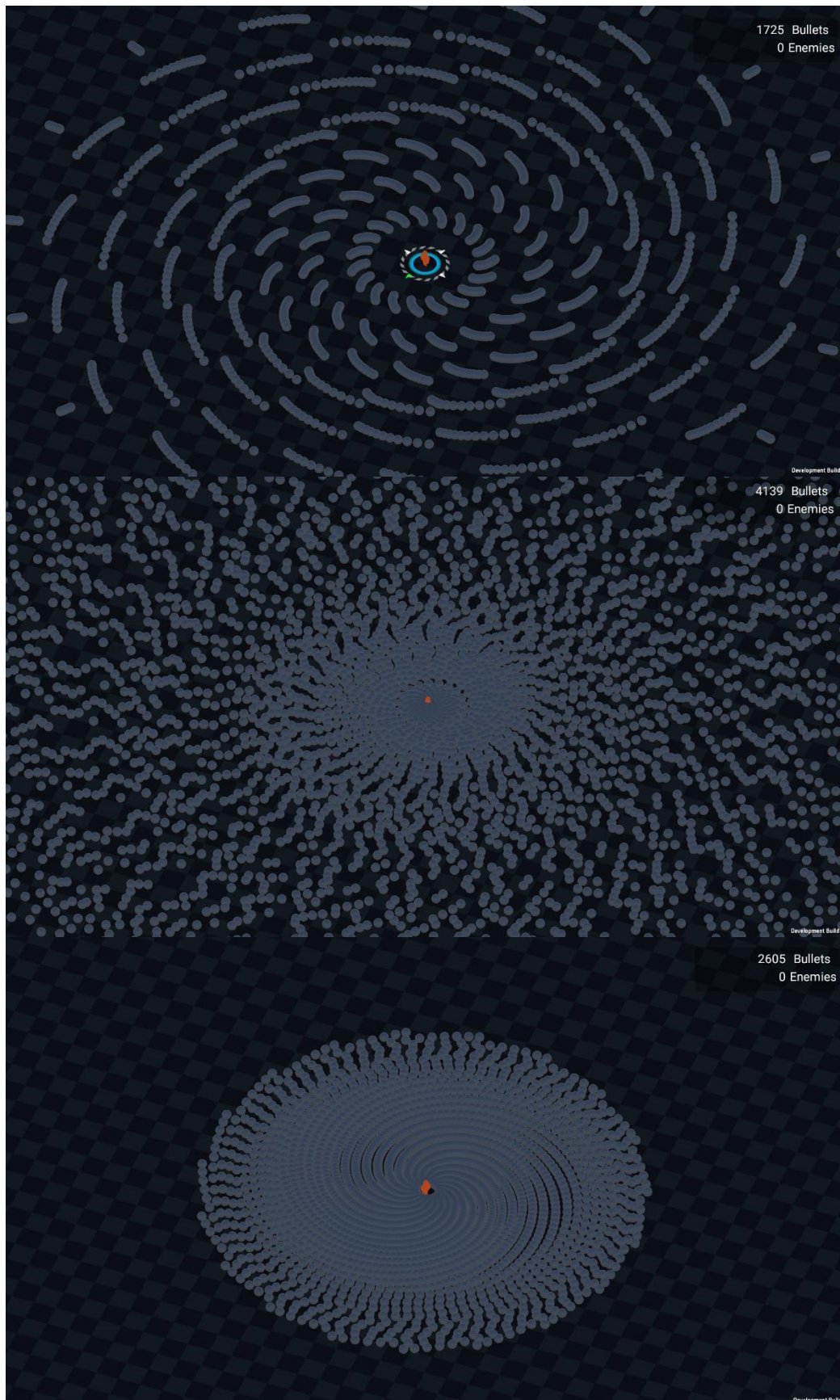


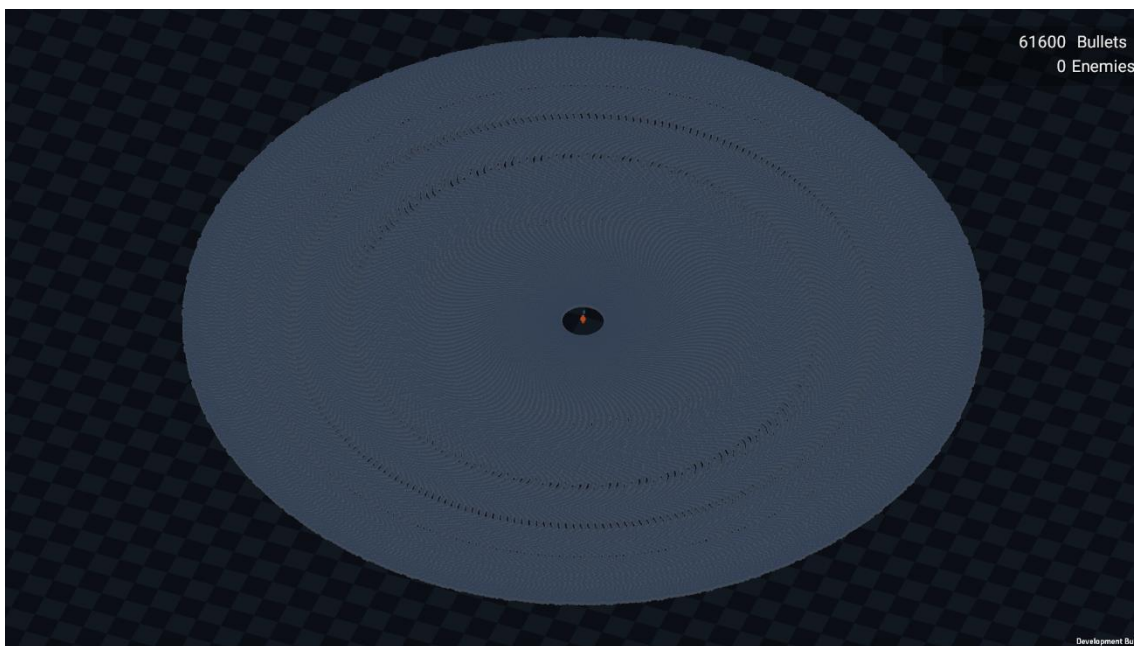
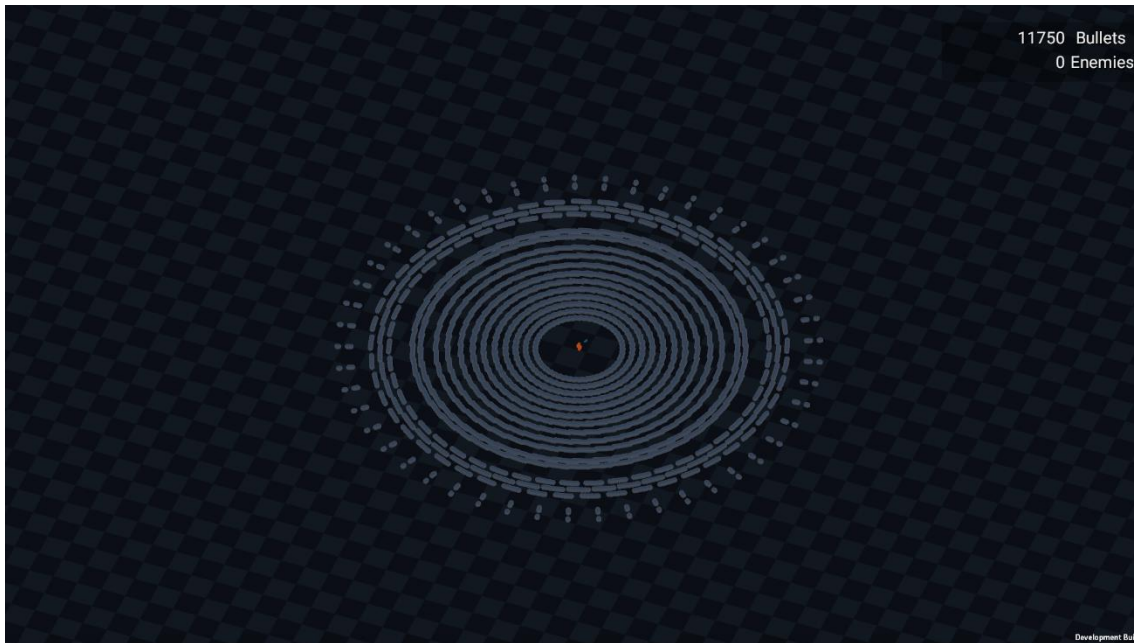
A high-resolution .svg file of the bubble map can be found at [this link](#) for a more in-depth look.



12.5. Annex E

12.5.1. Gallery





12.6. Annex F

12.6.1. Data Collection: Unity's profiler

Here, Unity's profiler can be seen running while the build generates the specific behavior to record, in this case, shooting. When 300 frames have passed since the goal was reached, the build is stopped, and calculations are taken from the profiler along 100 frames to get an average metric of the studied case. The first pair of images are from the Object Oriented Design, and the second pair is from the Data Oriented Design.

