Unity DOTS (Data Oriented Tech Stack) - Analyzing the technical relevance of the Data Oriented Tech Stack to the Unity Engine

Module Number:	CMN6302
Module Name:	Major Project (B. Sc.)
Date Submitted:	26.08.2022
Award Name:	Bachelor of Science (Hons)
	Games Programming
Course:	GPBP 0919
Semester:	Sommer Semester 2022
Name:	*Zayarmoe Kyaw*
City:	Stuttgart
Country:	Germany
Word Count:	14.369
	(13.092 + 1.277 direct citations)

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken (dazu zählen auch Internetquellen) entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht.

Stuttgart, 26.08.2022

Ort, Datum

Zayarmoe

Unterschrift Student

Table of Contents

Table of figure

Table of Tables

1 Introduction

1.1 Problem definition

For a games programmer there are a lot of choices when it comes to how to create a game. Starting with just the programming language there is already a huge selection ranging from assembly languages to high-level programming languages. But the most important choice for a game programmer is which game engine to use. There has been a lot of games engine throughout the years from the Source engine to the CryEngine. There are a lot of game engines currently available on the market. But when first introduced to game development, Unity or Unreal are the most commonly used game engines *(Toftedahl, 2019)*.
Unity and Unreal are certainly great choices but don't fully take advantage of the current state of technology. Technology has evolved from single core processors to multicore processors. Meanwhile game engines haven't evolved to utilize this change, so more and more newer video games struggle with performance on mid to low-end CPUs and require a lot more manual optimization by the developers.

Some of the technical feature that have become more prominent in the recent years are multithreading and the use of native code. Unfortunately, these features are difficult subjects to learn. Besides the poor utilization of multicore processing in Game Engines, the difference in memory speed compared to CPU processing speed has grown over the years like shown in figure 1.

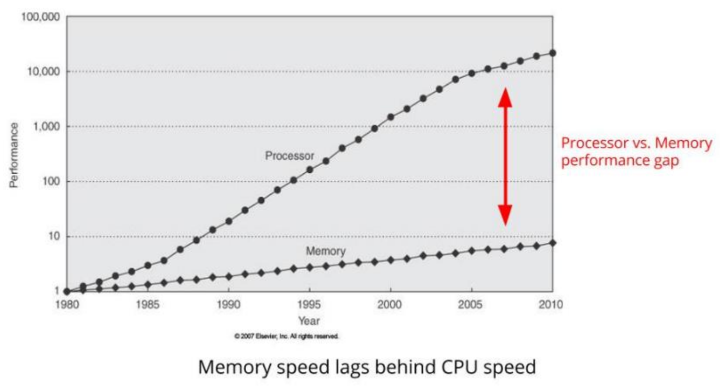


Figure 1 The difference between CPU clock speed and memory bus speed (Cheney, 2014)

So, at the Unity Unite conference in 2019 Unity Technologies introduced Unity DOTS and showed off impressive features that the new system DOTS brings to the Unity Engine *(Unity Technologies, 2019)*. Unity Technologies states that the new Data-Oriented Technology Stack (short DOTS) will make it possible to take full advantage of today's multicore processors without the heavy programming headache *(Unity Technologies, no date a)*. The core features of DOTS were already presented by Unity Technologies at the Unite 2018 with the Megacity demo. The Megacity demo contains 4.5 million mesh renderers, 5000 dynamic vehicles and 200,000 unique buildings *(Unity Technologies, 2018)*. The Megacity demo is certainly an impressive display of the capacities of Unity DOTS.
Even though Unity DOTS was introduced a few years ago, the information on developing games using DOTS is pretty scarce and mostly outdated because of its frequent updates.

1.2 Research Objectives

The objective of this thesis is to analyze and evaluate the relevance of the technologies introduced by Unity DOTS for the Unity Engine and the industry. For this purpose, the core features and most essential libraries of Unity DOTS are compared to the current Unity Engine under the aspects of performance.
In addition to the performance testing the structure of the new coding style introduced through the Entity Component System (ECS) will be analyzed under the aspect of usability in contrast to the current object-oriented style.

1.2.1 Target group

The target group for this thesis are Unity developers with a basic understanding of coding and junior programmers interested in game development with Unity. This thesis gives reader an introduction to software design in video games and the Data-Oriented Technology Stack (DOTS). But also aims to give the reader a technical evaluation of the software design and software optimization to determine whether to use the current Unity Engine or Unity DOTS.

1.2.2 Hypotheses / Problem and Research Questions

In the context of this thesis “classic Unity” refers to the standard system in an unmodified Unity project. In addition to the hypotheses of this thesis the statement made by Unity Technologies regarding Unity DOTS will be analyzed as hypotheses.

The statements made by Unity are as mentioned in the following citations.

> “DOTS will enable you to create richer user experiences and iterate faster with C# code that’s easier to read and reuse across other projects.” *(Unity Technologies, no date b)*
>
> “Our Data-Oriented Technology Stack (DOTS) will make it possible to take full advantage of today’s multicore processors without the heavy programming headache.” *(Unity Technologies, no date a)*

The hypotheses analyzed by this thesis are as mentioned in the following.

- A Unity DOTS project requires less memory during runtime.
- Unity DOTS code executes faster than classic Unity code.
- The hybrid renderer of Unity DOTS renders polygons more efficiently than the classic Unity renderer resulting in more fps (frames per second) by same polygon count.
- The ECS makes data-oriented design easier and more accessible with less effort.

The overall research question of this thesis is whether the architectural pattern entity component system is a solution or improvement to the performance and usability of the Unity Engine.

1.3 Relevance

An evaluation on the market share of game engines on the platform Steam shows that around 50% of games are developed using the Unity engine while games using the Unreal engine only make up around 13%. Unfortunately for the Unity Engine, “Unreal is favored for larger-scale projects […] [with] 25% of Unreal games launch at a price point of $29.99+, compared to only about 6% of Unity games." Additionally, only two Unity games have reach platinum top grossing on Steam since 2016 (see figure 2). Games, that were launched with a price less than $4.99 and have fewer than 50 reviews, were filtered out from the statistics results. (Doucet/Pecorella, 2021)

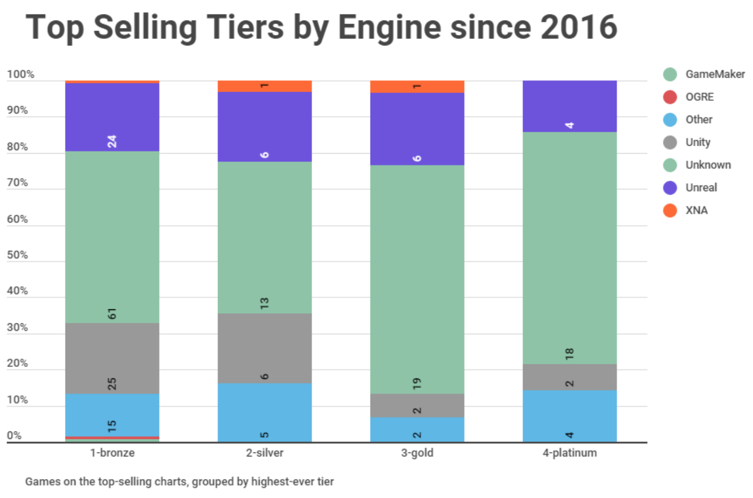


Figure 2 Top grossing games by engine on Steam since 2016 (Doucet/Pecorella, 2021)

With the data-oriented design introduced to the Unity Engine with DOTS the market position on PC could drastically change. The Unreal Engine makes competition with its upcoming release of Unreal Engine 5 in early 2022 (Unreal, no date).

With Unity not natively supporting data-oriented feature and utilizing C# over C++, its main demographic has mostly been Indie studios. The new technology introduced by Unity DOTS possess the capacity to overcome the disadvantages through the optimization of the internal architecture. In addition, Unity DOTS could make multithreading more accessible to smaller developers establishing a huge advantage over other engines.

As of writing this thesis comparison and information on developing using DOTS found were deprecated and of older version of DOTS. The results of this thesis should give the reader an impression of the current technical state of Unity DOTS and give an understanding of the benefits and disadvantages using Unity DOTS.

2 Context

2.1 Unity Data Oriented Technology Stack (DOTS)

As mentioned in the introduction of this thesis computing has been evolving over the last decade but the games industry hasn't fully adapted to this evolution yet. In regards of this evolution Unity Technologies has seen the need to design something that can fully utilize the current state of computing introducing Unity DOTS. The Data-Oriented Technology Stack (DOTS) is the name for Unity Technologies' attempt at rebuilding its internal architecture of the Unity Engine in a way that is faster, lighter, and, more important, optimized for the current multi-threading world. So, Unity DOTS is this thesis subject to analyze.
Unity DOTS has an extensive library but for the purpose of creating a solution to the multithreaded world, it introduces three main components. The components are the Entity Component System (short ECS) with a data-oriented design, the Unity C# Job System with an engine internal multithreading system and the Burst Compiler to further speed up processing.

2.2 Software design

Before talking about the data-oriented design introduced with the entity component system let's first establish the terminology used in software design. This allows for a better understanding of how the entity component system interacts with object-oriented programming and how it changes the structure of the code.

"[All began] when digital computers emerged in the 1950s, software was written in machine language." *(Garlan/Shaw, 1994, p.3).* Machine language requires the entire program to be manually checked for the insertion of new instructions. Eventually the process of memory layout and references was automated. With the automation resulted symbolic assemblers, which allowed commonly-used sequences to be called with a single symbol. In the late 1950s certain pattern that were commonly useful were better understood and automated. "The first of these patterns were for evaluation of arithmetic expressions, for procedure invocation, and for loops and conditional statements." *(Garlan/Shaw, 1994, p.3)*. These patterns lead to a series of early high-level programming languages. *(Garlan/Shaw, 1994, p.3).*

2.2.1 Programming paradigm

These patterns used in programming languages are called programming paradigms. Nørmark defines programming paradigms as “a pattern that serves as a school of thoughts for programming of computers” *(Nørmark, 2014).*
Paradigms can be considered as a collective set of programming concepts. Every programming language realizes one or more paradigms. Because there are many fewer paradigms than languages, multiple languages realize the same paradigms. From the viewpoint of paradigms languages such as Java, Javascript, C#, Ruby and Python are virtually identical. That why it’s more interesting to analyze the programming paradigms than the programming languages themselves. *(Van Roy, 2009, pp.10 and 12)*

> “Programming languages focus on data structure and algorithm of the computation [through the principles of a given paradigm] […]” *(Leavens, 1998).*

2.2.2 Design patterns

Another part of software design is the software architecture. The software architecture is divided into various layer of abstraction.
“[…] [On one of the lower levels of a software architecture] resides the architecture of the modules and their interconnections. This is the domain of design patterns […]” *(Martin, 2000, p.1)*

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" *(Alexander et. al., 1977, p. x, cited in Gamma et. al., 1995, p.2)*

This quotation talks about patterns in buildings and town, but it’s also applicable to design patterns in software design. Design patterns can be described as an established template for solving a common coding task in a generic way or as principles to solve certain problems. *(Gamma et al., 1995, p.2)*

Comparatively to programming paradigms that are predefined in the programming language, design patterns describe a pattern as a solution to a problem applicable in any programming language. Programming paradigms intend to layout the structure of the implementation, while design pattern intend to give a template to organize the software in order to solve a problem.

Design patterns are created over years of experience and provide less sophisticated developers with a pre-packaged solution to a common design problem. *(Garfinkel, 2005, pp.14-16)*

2.2.3 Architectural pattern

“At the highest level [of a software architecture], there are the architecture patterns that define the overall shape and structure of software applications” (Martin 2000)

Architectural patterns are design pattern, but what makes a design pattern an architectural pattern. “[A design pattern can be considered] […] architectural, if it refers to a problem at the architectural level of abstraction; that is, if the pattern covers the overall system structure and not only a few individual subsystems.” (Avgeriou/ Zdun, 2005).

Design pattern and programming paradigms are applied during the implementation phase of a program, while architectural pattern need to be applied during the design phase of a program.

2.3 Object-oriented programming

The Unity Engine scripting API is written in C#. “[The programming language] C# is intended to be a simple, modern, general-purpose, object-oriented programming language” *(ISO/IEC, 2018, p.xxi).* The object-oriented paradigm isn’t the only paradigm/principle realized in C# but for the purpose of this thesis the primarily relevant paradigm.

The object-oriented programming paradigm (short OOP) is a paradigm based on the concept of objects *(Holopainen, 2016).*

> “Object-oriented programs are made up of objects. An object packages both data and the procedures that operate on that data. The procedures are typically called methods or operations. An object performs an operation when it receives a request (or message) from a client.” *(Gamma et. al., 1995, p.11)*

The object-oriented paradigm follows major four principles to manage the software structure.

Encapsulation

“[…] [E]ncapsulation is the hiding of data [and] implementation by restricting access to accessors and mutators.” *(Lewallen, 2005)*

Abstraction

“[…] a simple definition of data abstraction is the development of classes, objects, types in terms of their interfaces and functionality, instead of their implementation details.” *(Lewallen, 2005)*

Inheritance

In some cases, two classes can share the same attributes and still be too distinct to be generalized into a single class. *(Holopainen, 2016)*

Instead of having each class implement the same functionality and characteristics, Inheritance allows one class to inherit them from another class. *(Lewallen, 2005)*

Polymorphism

“Polymorphism means one name, many forms. Polymorphism manifests itself by having multiple methods all with the same name, but slighty [*sic*] different functionality.” *(Lewallen, 2005)*

“Object Oriented Programming is a methodology, a design principle, programming paradigm, a concept, a practice, etc. It is not a language!” *(Roathe, 2010)*

2.3.1 Object-oriented design

Object-oriented design is the design pattern that generalizes the OOP’s principles for other languages. Object-oriented design (short OOD) and programming are very similar when looking at the principles that both define because both are still object oriented. The object-oriented design was created to counteract the mis practices when programming in OOP.

Object-oriented design has five principles for class design which are named *SOLID.*

The Single Responsibility Principle

“A class should have one, and only one, reason to change.” *(Martin, no date)*

The Open Closed Principle

“A module should be open for extension but closed for modification.” *(Martin, 2000)*

The Liskov Substitution Principle

“Subclasses should be substitutable for their base classes.” *(Martin, 2000)*

The Interface Segregation Principle

“Many client-specific interfaces are better than one general purpose interface.” *(Martin, 2000)*

The Dependency Inversion Principle

“Depend upon Abstractions. Do not depend upon concretions.” *(Martin, 2000)*

2.3.2 Composition over Inheritance

A common approach to implementing object orientation is the use of inheritance.

> "[…] [Unfortunately,] the traditional game object hierarchy ends up creating the type of object known as "the blob". The blob is a classic "anti-pattern" which manifests as a huge single class […] with a large amount of complex interwoven functionality." *(West, 2007)*

These "blobs" can result from overusing inheritance and using inheritance as the primary reuse mechanism. A false assumption in OOP is the need to use inheritance to achieve flexibility and reusability leading to overusing it *(Knoernschild, 2001, p.17 and 183)*. This resulted in an anti-pattern conflicting with OOD presented as huge classes that could handle every possible situation negating the purpose to ease code management *(Hollmann, 2019, p.7).*

> "The solutions [to this issue] […] can be categorized as composition-based architectures in which […] objects are represented as collection [or composition] of components. In these architectures inheritance hierarchies are either flat or non-existent." *(Hollmann, 2019, p.7)*

In this collection of components approach, the functionality of an object is separated into individual components that handle a single task and are mostly independent from each other. The classical object now becomes a collection of these individual components that can implement new functionality by adding a component. *(West, 2007)*

This principle is called composition over inheritance or composite reuse principle.

2.4 Data-oriented design

One of the core features introduced by Unity DOTS is the entity component system which implements data-oriented principles. To understand the purpose of using the entity component system, the principles of data-oriented design need to be defined.

As the name suggest this design pattern differs from the object focus and instead revolves around data*. (Llopis, 2009)*

More than just a design pattern, the data-oriented design (short DOD) is an optimization approach that "[…] carefully considering the memory layout of data structures, and their implications for auto-vectorization and use of the CPU cache." *(McMurray, 2020)*

2.4.1 CPU caching

Before talking about how the data-oriented design considers memory layout, the way the memory is read and written needs to be established.

CPU cache

CPU can process data much faster. Comparatively the memory can't get data fast enough. In order for memory to keep up the CPU has a small chunk of memory inside the chip called *cache*. The data stored in the cache can be pull much faster than from memory. Modern CPUs have multiple levels of caching representing different *cache lines*, which are called "L1", "L2", "L3" etc. Each level is larger but slower than the previous. *(Nystrom, no date)*

Cache accessing

When the CPU requests a byte of data from RAM, a whole chunk of memory gets loaded into the cache like seen in figure 3. The cache loads an entire vectorized chunk of memory even if only one byte is needed. If the next byte requested is within that chunk it's called a *cache hit*. If the next byte isn't within that chunk, it's called a *cache miss. (Nystrom, no date)*

When a cache miss occurs, the request continues through each level of cache until the requested byte is found and read from memory*. (Losonczi, 2020)*

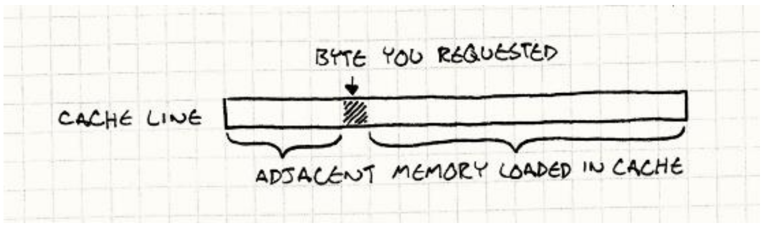


Figure 3 Caching of data in CPU when requesting data (Nystrom, no date)

2.4.2 Data locality

One of the data-oriented principles is to create ideal data which takes the cache into consideration. *Noel Llopis (2009)* presents the idea of ideal data which is data in a format that takes the least amount of effort to use. Effort in this case means the amount of processing or transformation of the data that is required.
To understand the ideal data format, take the example of how data would be laid out in object-oriented compared to the data-oriented by using the data structures Array of Structures (short AoS) and Structure of Arrays (short SoA).

In object-oriented approach the data is naturally structured in some sort of tree like for example an inheritance tree or reference calls*. (Llopis, 2009)*

> “As a result [of the tree structure], when we perform an operation on an object, it will usually result in that object in turn accessing other objects further down in the tree. Iterating over a set of objects performing the same operation generates cascading, totally different operations at each object […] [(see figure 4)]”. *(Llopis, 2009)*

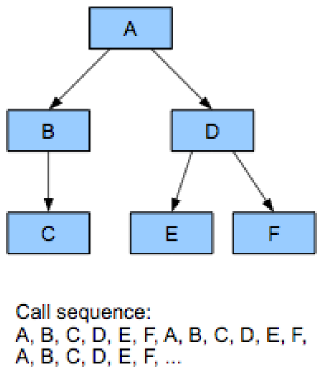
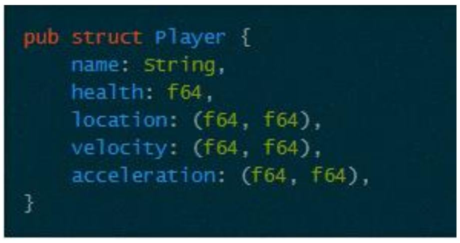


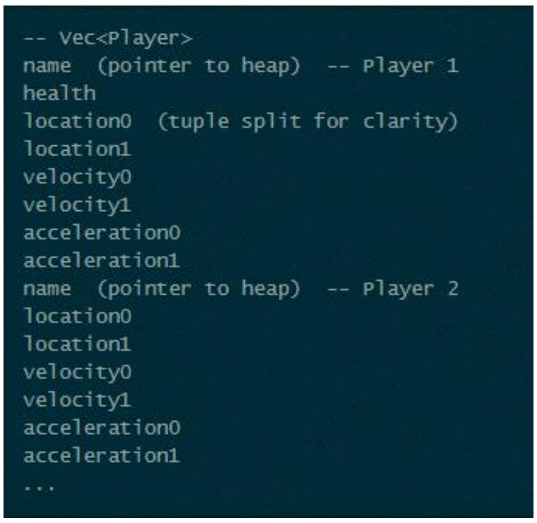
Figure 4 Call sequence with an object-oriented approach (Llopis, 2009)

An implementation that represents this tree structure would be an array of structures (AoS). Take the example of creating players that contain their position and other variables. In an AoS implementation, an array is created containing structs. Figure 5 shows an example for a player struct and figure 6 presents the memory layout of an array containing players. *(McMurray, 2020)*



```
pub struct Player {
    name: String,
    health: f64,
    location: (f64, f64),
    velocity: (f64, f64),
    acceleration: (f64, f64),
}
```

Figure 5 Player struct in AoS (McMurray, 2020)



```
-- Vec<Player>
name  (pointer to heap)  -- Player 1
health
location0  (tuple split for clarity)
location1
velocity0
velocity1
acceleration0
acceleration1
name  (pointer to heap)  -- Player 2
location0
location1
velocity0
velocity1
acceleration0
acceleration1
...
```

Figure 6 AoS Player data represented in memory (McMurray, 2020)

In this approach the cache needs to load an entire player when the code desires to change a field because of its intertwined structure like seen in figure 7. When dealing with multiple objects the cache generates a lot of cache misses which slows the program drastically.

> "OOP deals with each object in isolation" *(Llopis, 2009)*

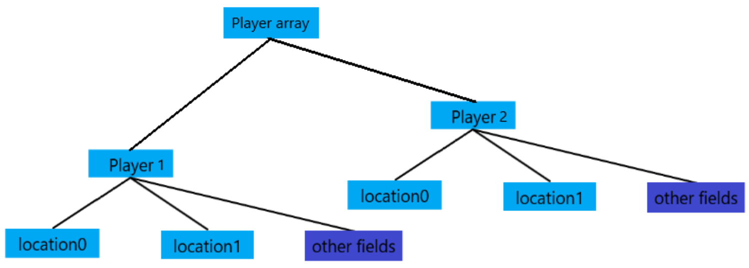


Figure 7 AoS player array presented as a tree (Kyaw, 2021)

In data-oriented approach to create the ideal data, each object is broken down to its individual components and same types of components are grouped together regardless of their object. *(Llopis, 2009)*

> "This organization results in large blocks of homogeneous data, which allow us to process the data sequentially […] [(see figure 8)]". *(Llopis, 2009)*

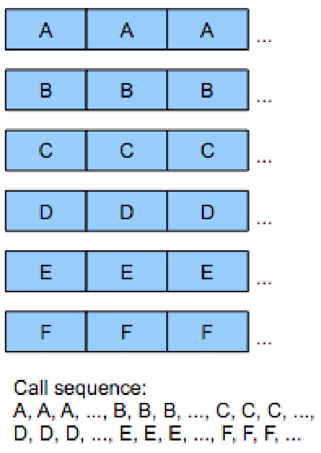
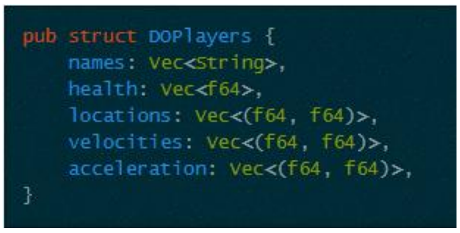

A A A ...
B B B ...
C C C ...
D D D ...
E E E ...
F F F ...
Call sequence:
A, A, A, ..., B, B, B, ..., C, C, C, ...,
D, D, D, ..., E, E, E, ..., F, F, F, ...

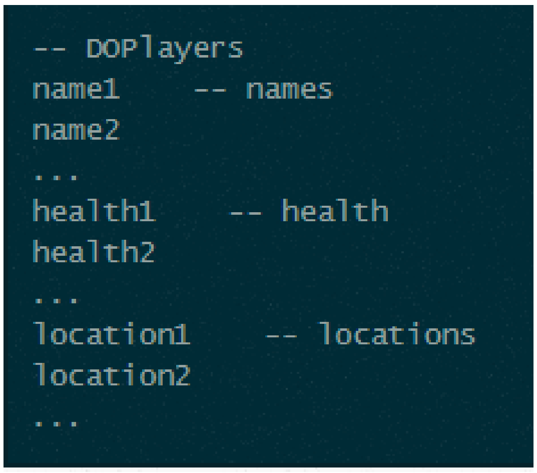
Figure 8 Call sequence with a data-oriented approach (Llopis, 2009)

An implementation that represents this ideal data structure would be a structure of arrays (SoA). In a SoA implementation, a struct is created containing arrays. Figure 9 shows an example for an array containing player fields as arrays with each index representing a player and figure 10 presents the memory layout of that array. *(McMurray, 2020)*



```
pub struct DOPlayers {
    names: Vec<String>,
    health: Vec<f64>,
    locations: Vec<(f64, f64)>,
    velocities: Vec<(f64, f64)>,
    acceleration: Vec<(f64, f64)>,
}
```

Figure 9 Player struct in SoA (McMurray, 2020)



```
-- DOPlayers
name1     -- names
name2
...
health1    -- health
health2
...
location1    -- locations
location2
...
```

Figure 10 SoA Player data represented in memory (McMurray, 2020)

In contrast to AoS, SoA doesn't need to load an entire player and can instead load the location data to update directly. The location of the player doesn't have a direct connection to the other fields of the player. Because the positions are loaded into memory next to each other, the next location to update results in a cache hit.
“[This approach] […] [takes] advantage of […] [the cache] to improve performance by increasing data locality—keeping data in contiguous memory in the order that you process it.” *(Nystrom, no date)*

Contiguous memory describes memory that is structured like show in the concept of ideal data. Figure 11 show the visual representation of contiguous memory compared to non-contiguous memory. Ideal data and contiguous memory are defined by the principle called data locality or locality of reference.

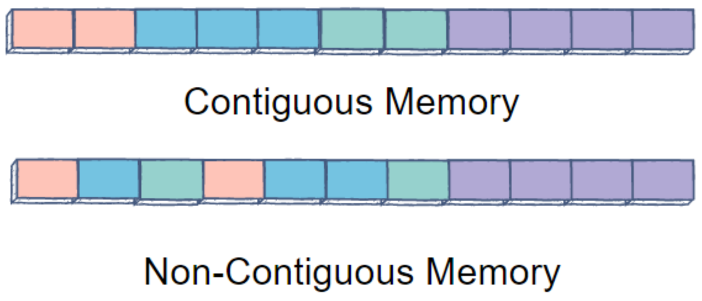


Figure 11 Visual definition of contiguous memory (Edpresso, no date)

2.4.3 Separate data from logic

Another principle of data-oriented design is the separation of data from logic which works hand in hand with the data locality. In an object-oriented approach, the code contains logical entities with fields of everything that logically belongs to that object with its behavior (see figure 12 bottom). Instead of having object that contain data and logic, in a data-oriented approach they're separated in regards of its usage. Additionally, the data is completely separated from the system that operates on it (see figure 12 top). *(Nikolov, 2018)*

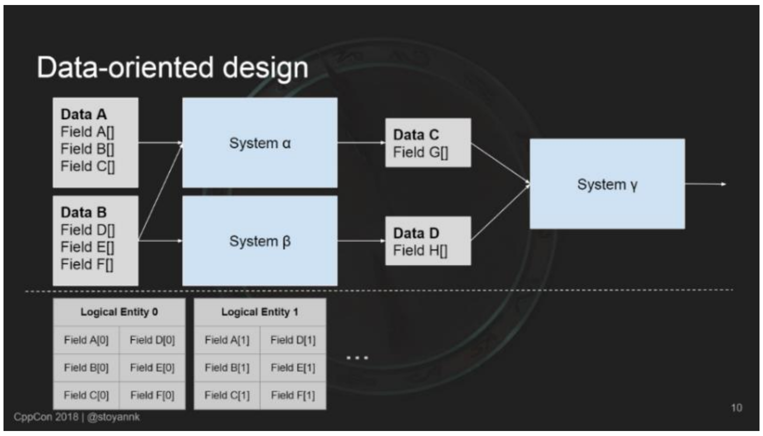


Figure 12 data layout in DoD (top), data layout in OOD (bottom) (Nikolov, 2018)

2.5 Entity Component System (ECS)

With all related principles defined, the primary change to the Unity Engine introduced by Unity DOTS can be explained. Unity DOTS introduces the architectural pattern Entity-Component-System (short ECS) to the Unity Engine.

> "An Entity Component System (ECS) architecture separates identity (entities), data (components), and behavior (systems). The architecture focuses on the data. Systems read streams of component data, and then transform the data from an input state to an output state, which entities then index." *(Unity Documentation, no date a)*

The idea of separating the structure of a game logic into entity, component and system is pretty old. One of the earliest introductions of this pattern was by Scott Bilas in his GDC talk in 2002 on the development of Dungeon Siege. He states some of the common problems that arise with the conventional gameobject design such as the inflexibility of the code to change and the high complexity of the class trees. *(Bilas, 2002)*

The ECS is an architectural pattern that enforces the principles given by the object-oriented design by using principles of the data-oriented design. Some of the applied principles are the composition over inheritance and data locality and separation of data from logic from DOD.

2.5.1 Entities and Components

The first element of an Entity Component System is the *Entity*. "An entity has neither behavior nor data; instead, it identifies which pieces of data belong together." *(Unity Documentation, no date b)*
"Entities identify sets of components to represent simulation objects" *(Hollmann, 2019).* "[…] an entity is little more than a list of components. […] an entity is a unique ID, and all components that make up an entity will be tagged with that ID." *(Klutzershy, 2013)*

The second element of the ECS are the *Components*. Components have no behavior implemented and are only capable of storing data. Each component only describes an aspect of an entity. *(Klutzershy, 2013)*
"Components are simple structs of plain old data" *(Hollmann, 2019, p.10)*
The ECS applies the principle of composition over inheritance using these components to solve the memory problem described in 2.4.2 Data locality with inheritance trees. For the system to process the data from the components there is no encapsulation

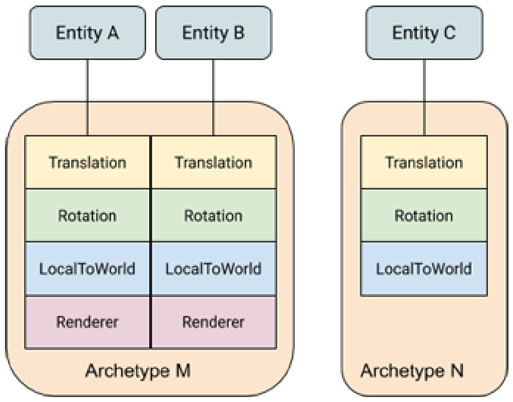


Figure 13 Visual representation of the Entity archetypes (Unity Documentation, no date a)

The unique compositions of components are grouped as an *EntityArchetype* (see figure 13). Figure 13 show two different Archetype M and N and the entities A, B and C which identifies the corresponding components to the entity.
A similar concept to the composition-based structure of ECS already exists in the classic Unity with the inspector. Adding components like Transforms and Rigidbody follow the same principle as the ECS components.

The ECS applies data locality using the SoA structure on the EntityArchetypes by packing entities of the same archetype in the same memory chunk (see figure 14) *(Hollmann, 2019, p.10)*. The memory chunks are called *ArchetypeChunks*. A chunk will only contain entities of the same archetype. When the composition of an entity changes the ECS allocates a new chunk that fits that archetype. This structure allows for faster iterating through entities by searching for a certain composition of components. *(Unity Documentation, no date a)*

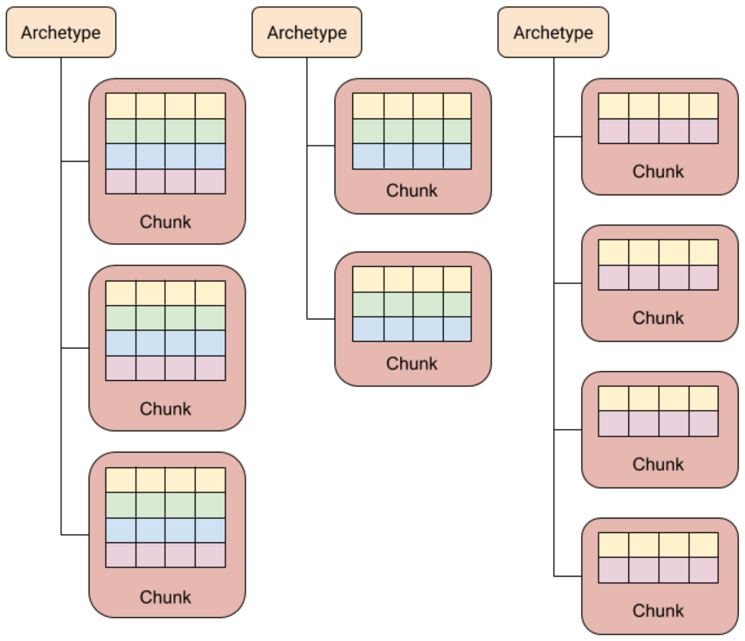


Figure 14 Visual representation of the memory allocation of entities with archetypes (Unity Documentation, no date a)

2.5.2 Systems

The third element of the ECS are the *Systems* which represent the behavior.
The behavior implemented using system works fundamentally different to OOP. In traditional OOP each object defines its own behavior which results in wasted memory when there are multiple instances of a class with each defining the same method. In addition, methods need to be invoked externally. *(Adam, 2007)*
On the contrary in ECS each objects/component possess no behavior instead an external system operates on a certain composition of components. *(Adam, 2007)*

> "[In ECS] [e]ach System runs continuously (as though each System had it's [*sic*] own private thread) and performs global actions on every Entity that possesses a Component of the same aspect as that System." *(Adam, 2007)*

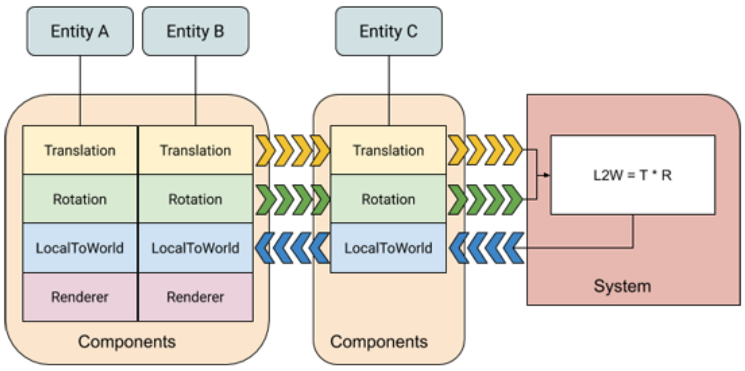


Figure 15 Visual representation of the Unity ECS architecture (Unity Documentation, no date a)

Figure 15 show the system existing independently from the data that it operates on. The system in figure 15 operates on entities with the composition of Translation, Rotation and LocalToWorld. The Renderer component in this example is ignored because of its irrelevance to the calculation of the system.

> “Each system has a narrow focus on a few aspects of behavior [applying the “Single Responsibility Principle”] and iterates over the relevant component-arrays[/EntityArchetypes] only” *(Hollmann, 2019, p.10)*.

2.5.3 Unity’s ECS

In addition to the general structure of an ECS Unity DOTS implements additional functionality like entity conversion, hybrid renderer and hybrid components.

The *entity conversion* feature converts gameobjects into entities by translating the classic Unity components into ECS equivalent components. For components that do not have ECS versions yet, hybrid components are used. Hybrid components allow for ECS code to access classic Unity components. This feature only converts data into ECS and does not convert the implemented behavior.

The hybrid renderer is the Unity DOTS renderer that allows Entities to be rendered. For an entity to be render it requires the following DOTS components: LocalToWorld, RenderMesh and RenderBounds.

2.6 Multithreading

As mentioned in the introduction the processing power of CPUs has evolved from single core processors to multicore processors. Simultaneously the amount of data utilizes in programs has drastically grown. It became essential for data to be processed faster resulting in something called parallel programming. Parallel programming describes the process of executing operations concurrently. *(Subburaj, 2020)*
There are multiple approaches to parallel programming such as multiprocessing, multitasking and multithreading. In regards to this thesis multithreading is the relevant approach.

Preliminarily let’s look at some technical term before talking about multithreading.
“[A program] […] is a collection of instructions, an executable file, [..] that performs a specific task when executed.” *(Subburaj, 2020)*
“[A process] […] is a running instance of a program and is always stored in the RAM, each process has its own memory, data and other resources needed to execute the program.” *(Subburaj, 2020)*
“[A thread] […] is an entity that resides within a process, every process is started with a single thread called the primary thread. […] One or more threads can be spawned within a process and all of them share the same memory space allocated to the process and get executed within the scope of a process.” *(Subburaj, 2020)*

Multithreading is an approach where multiple threads are created within a single process to assign operations to them so that they can be executed concurrently to the main thread. All these threads run in the same memory space as the main thread. *(Subburaj, 2020)*

2.6.1 Synchronizing in multithreading

With all threads sharing the same memory space synchronizing between them is required to “[…] ensures that two or more concurrent threads do not simultaneously execute some particular program segment known as [the] *critical* *section*.” *(GeeksforGeeks, 2019)* “Critical section refers to the parts of the program where the shared resource is accessed.” *(GeeksforGeeks, 2019)*

When multiple threads access a critical section and attempt to change its values at the same time an issue called *race condition* occurs. This issue makes the values of the object unpredictable and dependent on the timing of the context switching of the process. *(GeeksforGeeks, 2019)*

The figure below presents a simple depiction of a race condition. Each thread increments the integer x by one. The expected result at the end of the graph is 12 but due to race condition the second thread does not wait for the completion of thread one resulting in 11.

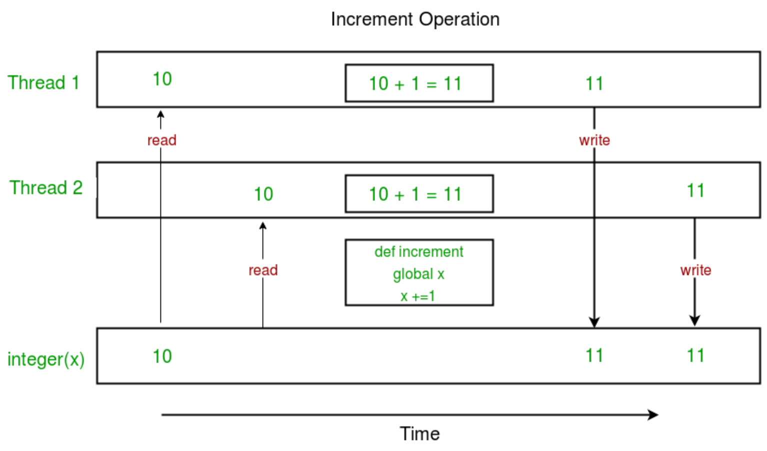


Figure 16 Race condition example (GeeksforGeeks, 2019)

2.6.2 Solution to race condition - Locks

In order to deal with race conditions threading libraries provide the use of locks.

The figure below presents a simple depiction on how locks work. When thread 2 accesses the integer x, it places a lock on it preventing thread 1 from accessing it until its finished using it. After thread 2 is finished, it releases the lock on x and allows thread 1 to access it.

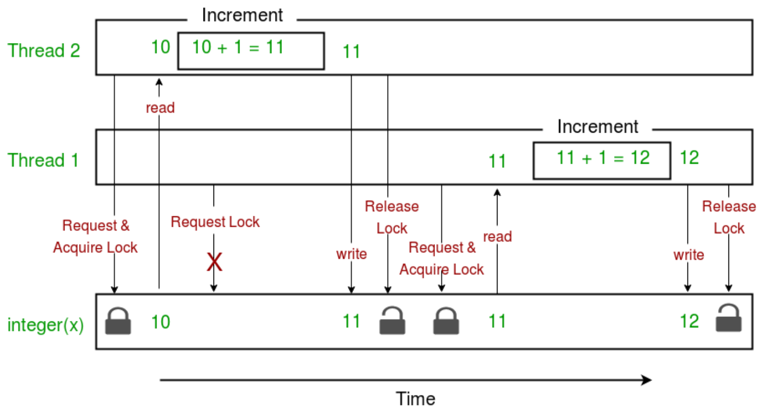


Figure 17 Locks use case example (GeeksforGeeks, 2019)

Locks is certainly a good solution to prevent race conditions but not without flaw. With the use of lock another issue can arise which is called a *deadlock*.

> "Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in a different order" *(Sam, 2020)*
>
> "[For example] a deadlock […] [would occur] when two threads each lock a different variable at the same time and then try to lock the variable that the other thread already. As a result, each thread stops executing and waits for the other thread to release the variable. Because each thread is holding the variable that the other thread wants, nothing occurs, and the threads remain deadlocked." *(Yu, 2022)*

The figure below presents a visual example of a deadlock occurring.

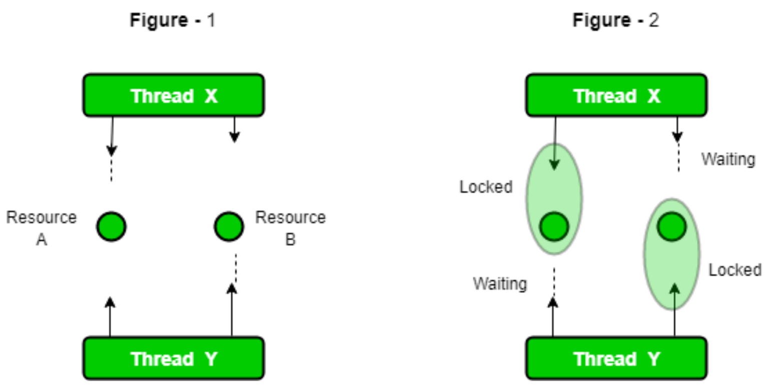


Figure 18 Visual example of a deadlock (GeeksforGeeks, 2021)

2.7 Unity's multithreading solution: Job System

As seen by the problems stated in 2.6 Multithreading, multithreading is not an easy feature to implement. So as stated by Unity Technologies *(no date a)*, DOTS "will make it possible to take full advantage of today's multicore processors without the heavy programming headache." The major feature introduced by Unity DOTS to achieve that is the multithreading solution C# Job system which is integrated into the engine.

The Unity Job system does not work like conventional multithreading. The Job system does not create threads to run certain operations instead it creates *jobs* which contains small units of work. These jobs are then scheduled on worker threads which are the same as the Unity engine internal job system. Using this approach ensures that there is not more than one thread per CPU core to prevent context switching*. (Johansson, 2018) (Unity Documentation, 2018)*

2.7.1 Safety system in the C# Job System

The engine integrated multithreading library offers various more features to prevent common multithreading issues such as race conditions and deadlocks.

In conventional multithreading as explained in 2.6 Multithreading the threads share the same memory space as the main thread. This aspect results in potential race conditions because of the dependency on timing between processes. Unity's job system completely eliminated the race condition issue by allowing job to only access a copy of the data it needs to operate on *(Johansson, 2018)*.

2.7.2 Native containers

> "The drawback to the safety system's process of copying data is that it also isolates the results of a job within each copy. To overcome this limitation, you need to store the results in a type of shared memory called NativeContainer." *(Unity Documentation, 2018)*

“A NativeContainer is a managed value type that provides a safe C# wrapper for native memory. It contains a pointer to an unmanaged allocation.” *(Unity Documentation, 2018)* Native containers need to be manually disposed similar to other native code like in C++.

These containers are given to the scheduled jobs as pointers to a copy of the data required and allow for its ownership to be transferred similar to the locks provided by multithreading libraries. To prevent deadlock, jobs require to be scheduled with a dependency to a previous job that accesses the container. Jobs will wait for the previous job to finish before executing. *(Unity Documentation, 2018)*

2.8 Unity Burst Compiler

The last major package that Unity DOTS introduces is the Burst compiler. “Burst is a compiler, […] [that] translates from IL/.NET bytecode to highly optimized native code using LLVM.” *(Unity Documentation, no date d).* The primary objective of the compiler is to improve the job system by compiling the jobs into native code pre runtime instead of on real time. *(Unity Documentation, no date d)*

Another use of the Burst compiler is the improvement of the Unity mathematics library by compiling the C# mathematics into highly efficient native code. Additionally, the new Unity mathematics library provides vector types and math functions with a shader like syntax like float4, float3, etc. *(Unity Documentation, no date e)*

3 Methodology

The hypotheses of this thesis analyze Unity DOTS under two aspects. The primary aspect is the usability of the new technology in comparison to the Unity Engine. The secondary aspect is the raw performance difference between the two environments, Unity DOTS and Unity Engine. To verify the hypotheses there are two approaches, gathering primary data and using secondary data.

In order to analyze the aspect of usability gathering secondary isn’t sufficient and hinders a personal evaluation. In addition, most secondary data don’t follow a certain standard of evaluating usability.
For analyzing the aspect of performance gathering data through conducting experiments provides results with a hardware context. Additionally, it also offers experience with the software to properly evaluate the usability.

With Unity DOTS being an unfinished beta system, the information and secondary data often uses older versions of Unity DOTS. Conclusively, gathering primary information through experiments allows for an accurate assessment of the current state of the new software. All experiments are subject to change depending on the newest version of the Unity DOTS packages.

The experiments are conducted as small-scale experiments instead of a game to ensure that all technologies and packages of importance can be analyzed thoroughly within the given time span. The experiments will be conducted on the same hardware to provide a context for the data gathered. The evaluation of the usability will be conducted under standard ISO 9241-11:2018.
ISO 9241-11:2018 defines usability as “the extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use” *(ISO, 2018).*

3.1 Project Setup

Before a project can use the Unity DOTS technology, certain packages need to be imported first. The following packages need to be imported for the Unity DOTS system to work:

- com.unity.entities
- com.unity.dots.editor
- com.unity.rendering.hybrid
- com.unity.physics

The packages are installed using the “Add packages from git URL...:” of the package manager. For a more detailed installation guide refer to the Unity Entities project setup guide.
(https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/install_setup.html)

These packages are available starting from Unity 2020.1. Some of the packages are also available in Unity 2019.1 but these versions are no longer in development and deprecated. So, this project will be using the LTS (Long Term Support) version of Unity 2020.3. The packages needed is subject to change because Unity DOTS is still under development. The version of packages used in this thesis will be included in the appendix.

3.2 Additional Tools

The project utilizes unity plugins and external software to more accurately monitor the performance during the experiments.
To monitor the memory allocation the Unity package Memory profiler is used.
For analyzing the rendering state, the external software Renderdoc is used.
How the programs are used for the experiments, is clarified in the experiments themselves.

3.3 Experiment Conditions

3.3.1 Hardware Specifications

The experiments are conducted on a laptop with the following specifications:
-CPU: Intel® Core™ i7-7700HQ CPU @ 2.80 GHz – 3.5GHz
- 4 Cores, 8 Threads
-GPU: NVIDIA GeForce GTX 1060 6GB
-Memory: 16GB 2400MHz
Because of the inconsistency of performance on laptop hardware, the results may differ to other devices with the same specifications.

3.3.2 Hardware Benchmark

For future reference, the hardware used for the experiments is going to be benchmarked. The benchmarks allow other people to replicate the experiments and get relative results to this thesis.
The benchmarks for the CPU and GPU are going to be conducted using *3DMark*. Additionally, for the CPU *Intel(R) Extreme Tuning Utility* will also be used. For the memory *UserBenchmark* will be used.

3.4 Experiment A: Memory allocation in Unity DOTS

Experiment A focuses on one of the main differences introduced by DOTS, the Entity Component System (short ECS) and analyses the memory allocation size difference resulting from using DOTS data types and components. More specifically the Entities created by the ECS. One of the hypotheses states that a game using Unity DOTS requires less memory during runtime compared to classic Unity. To answer this hypothesis a base line memory size difference between classic Unity and Unity DOTS data types and system will be determined. To test this hypothesis this experiment will be conducted in two sections.

3.4.1 Experiment A-1: Memory difference between data types

The first section of experiment A will compare the memory allocation difference between classic Unity data types and components to the Unity DOTS equivalents.
This process is done by checking the memory allocated of single initialized instances of the data types and comparing them to their equivalent types. For components added by the Unity Physics package the same process as the comparison of data types will be conducted. For container data types like lists and arrays, equally sized containers will be initialized containing equal amounts of the same items.
The data types with their respective comparison type can be found in the chart below.

Classic Unity	**Unity DOTS**
Unity mathematics	
Vector2/Vector2Int	float2/int2
Matrix4x4	float4x4
Quaternion	quaternion
Unity Physics	
Rigidbody	Physics Body
Collider	Physics Shape
Containers types	
List<N>	Native List
Array []	Native Array

Table 1 Unity component (left) with their corresponding DOTS component (right) for memory allocation size analysis (Kyaw, 2021)

3.4.2 Experiment A-2: Memory allocation using Entities

To verify the hypothesis, the second section of the experiment will analyze the memory behavior of gameobjects to their entity variants. This experiment will create three different types of objects/entities. The types to analyze are a gameobject, the entity converted version of that gameobject and a pure ECS entity.

This experiment will declare various fields in the objects/entities and compare its memory allocation size and overhead introduced by the monobehaviour class and component class. Every object and entity will declare the same fields for comparable results. For this comparison the declaration of functions and behavior in the gameobject is prohibited.

In addition, the memory allocation size of unity components when created as hybrid components will be analyzed too. This allows to determine whether the current state of unity dots could offer disadvantages because of its incompletion.

For this experiment the plugin memory profiler is going to be used. This plugin can analyze the memory used by specific field and objects, and gives information about the size, address, name and more about the allocated memory.

3.5 Experiment B: Runtime performance

Experiment B focuses on one of the bigger aspects when considering performance of a software, the speed at which data is accessed and created. One of the hypotheses states that code written in Unity DOTS executes faster than classic Unity code. This experiment will be conducted in two sections.

3.5.1 Experiment B.1: Read & write speed in DOTS

With DOTS comes a strong emphasize on the data-oriented design through the use of the ECS architecture. This section of the experiment will analyze the random read write speed of ECS code comparatively to classic Unity code.

For this analysis, a list of ten objects containing a single int field is created which is then accessed and written to randomly. This experiment will be designed as a stress test accessing the objects in batches of 1000+ times to determine the average time required to iterate through a set of data. This experiment will contain two different modes with read-write and readonly. This implementation procedure is realized in ECS and classic Unity to get a contrast between each version.

In order to account for data locality stated in chapter 2.4.2, this experiment will be conducted with two lists implemented using the data structure AoS and SoA. The purpose of this addition is the analysis of data-oriented data structure and whether small changes such as these can impact the code in any significant way.

3.5.2 Experiment B.2: Entity runtime instantiation

Unity DOTS offers various method of instantiating entities. For an assessment of the runtime performance of Unity DOTS the creating of entities hold relevance. This experiment will conduct load tests on the creation of Entities and Gameobject. During the process of the load tests the declaration of functions and behavior in the created gameobjects are prohibited.

During this experiment there will be three kinds of creation methods to account for. The methods are creating a gameobject normally, converting that gameobject to an entity and creating a pure ECS entity.
During the conversion process there will be two different approaches. The first approach converts the gameobject into an entity and instances more entities using that entity. The second approach converts the gameobject to an entity each time an entity is created to account for a scenario where multiple different gameobject are converted during runtime.
During this experiment, the declaration of functions and behavior is prohibited because the creation of behavior between classic Unity and ECS is not comparable.

The procedure of this experiment is similar to experiment A-2 in creating gameobject and entities with identical data stored in them and then monitoring the time required for the object/entity to be fully instantiated. Due to the low load times on instantiation, the experiment will be designed as a stress test creating 100+ objects/entities and determining the average time for a single instance.

3.6 Experiment C: Runtime speed and render capacity of DOTS

With DOTS using entities over gameobject, the way they are rendered is vastly different from one another. One of the hypotheses states that the hybrid renderer used by Unity DOTS renders polygons more efficiently resulting in a higher fps (frames per second) count by same polygon count.

For this experiment entities and gameobjects composed of mesh and material are rendered in large quantities ranging from 1000 to 100.000+ and instantiated within the render frustrum of the main camera in classic Unity and Unity DOTS. Through the use of the Unity profiler the number of polygons rendered by the camera in a certain frame is determined and put into correlation with the average fps.

3.7 Experiment D: Multithreading: Sorting algorithms

One of the major packages that DOTS introduces is its multithreading solution, Job System. Unity Technologies (no date a) states that DOTS allows “to take full advantage of today’s multicore processors without the heavy programming headache”. This experiment will analyze the usability of Unity DOTS’s multithreading solution, C# Job system.

To analyze the multithreading solution, this experiment will implement the sorting algorithm merge sort that has an average time complexity of $O(n\,log\,n)$ in different environments. The environments chosen are classic Unity without multithreading, DOTS without multithreading and DOTS with multithreading. This experiment will primarily focus on the usability aspect of the multithreading solution.
The merge sort algorithm is going to sort randomly constructed arrays of numbers of varying size.

3.8 Experiment E: Stress test of DOTS

The final experiment conducted will be a stress test that utilizes all previously tested aspect of DOTS. This experiment allows for a simulation of a high taxing scenario of a game environment.

For this experiment a large quantity of cubes with random position within the viewport of the camera are instantiated. The positions of these cubes are updated every frame by moving them to the opposite direction relative to the camera.
The instantiation of the cubes will be implemented in classic Unity, as an ECS conversion and as pure ECS. The movement behavior of the cubes will be implemented in classic Unity, in ECS, in multithreaded ECS and in multithreaded ECS enhanced with Burst.

4 Implementation

4.1 Project setup

The experiments are implemented and executed in a Unity project using 2020.3 LTS. To use the newest available Unity DOTS packages a Unity project in version 2020.3 LTS or newer is required.

The packages utilized differ from the packages mentioned in chapter *3.1 Project setup*. During the process of writing this thesis and the implementation of the experiments the Entities package and other ECS-based packages received an update to the 0.50 version (Fuad, 2022). The changes introduced by the updates effect the implementation of some experiments and offers improvement to usability and potentially performance. The installation process is as described in chapter *3.1 Project setup* excluding the “com.unity.dots.editor” package. Notable changes added by the new version of the Unity DOTS packages are mentioned within the experiment that are affected.

For a breakdown of the highlights of the update refer to Matt Fuad forum post. (https://forum.unity.com/threads/experimental-entities-0-50-is-available.1253394/). For a more detail list of changes provided by the new version of Unity DOTS refer to the changelog of the Entities package. (https://docs.unity3d.com/Packages/com.unity.entities@0.50/changelog/CHANGELOG.html).

In addition to the DOTS packages the memory profiler package is also added through the package manager. A detailed list of all packages used within this thesis is included in the appendix.

4.2 Stress test assets

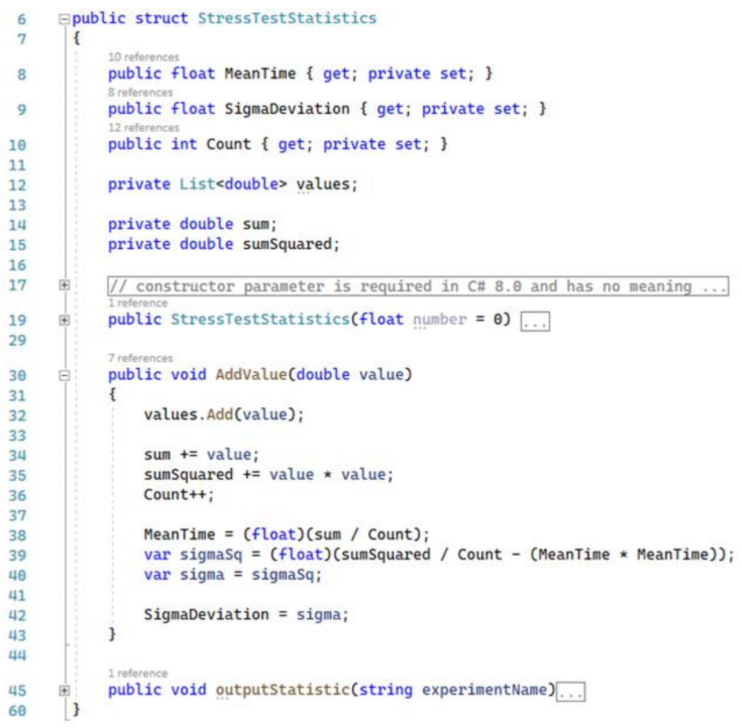
During Experiment B and E, the time required to process and create data is monitored. In order to monitor the time, the project implemented a descriptive statistic including the parameter mean time, standard deviation and count.

“Mean is the average level observed in some piece of data, while standard deviation describes the variance, or how dispersed the data observed in that variable is distributed around its mean." (Hayes, 2022)

A low standard deviation means that values tend to be closer to the mean while a high standard deviation means that values are further spread from the mean.

The descriptive statistic is very beneficial for the analysis of the hypotheses by allowing to determine the average time and the spread of the results. The average performance holds more value than determining the best and worst case for analyzing the performance between both versions.

The figure below shows the implementation of the descriptive statistic utilized.



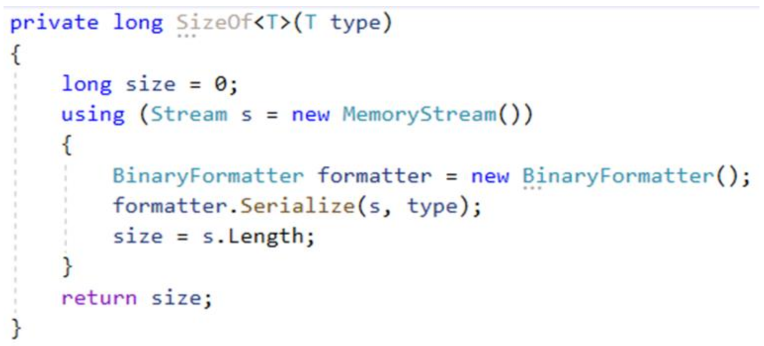
```csharp
public struct StressTestStatistics
{
    10 references
    public float MeanTime { get; private set; }
    8 references
    public float SigmaDeviation { get; private set; }
    12 references
    public int Count { get; private set; }

    private List<double> values;

    private double sum;
    private double sumSquared;

    // constructor parameter is required in C# 8.0 and has no meaning ...
    1 reference
    public StressTestStatistics(float number = 0) ...

    7 references
    public void AddValue(double value)
    {
        values.Add(value);

        sum += value;
        sumSquared += value * value;
        Count++;

        MeanTime = (float)(sum / Count);
        var sigmaSq = (float)(sumSquared / Count - (MeanTime * MeanTime));
        var sigma = sigmaSq;

        SigmaDeviation = sigma;
    }

    1 reference
    public void outputStatistic(string experimentName)...
}
```

Figure 19 Implementation of a descriptive statistics (Kyaw, 2021)

4.3 Implementation: Experiment A.1

This experiment analyzed the memory allocation size of data types in classic Unity and DOTS. The results of this experiment were gathered in bytes putting classic Unity component into relation with ECS components.

The first approach tried to implement the experiment as closely to described in 3.4.1 by creating objects/instances of the data types and using a binaryformatter. The binaryformatter serialized the instance of the data type to binary and allows to output the memory size of the created binary (see figure 20).

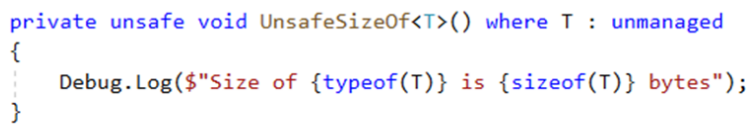


```
private long SizeOf<T>(T type)
{
    long size = 0;
    using (Stream s = new MemoryStream())
    {
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Serialize(s, type);
        size = s.Length;
    }
    return size;
}
```

Figure 20 Experiment A.1 first approach (Kyaw, 2021)

The issue with the approach was the binaryformatter serialized the object’s type info and value in result creating extra bytes. These extra bytes allocated by the binaryformatter represent the type information and are used for deserializing the object. The binaryformatter didn’t offer a solution to determine the extra number of bytes additionally allocated therefore a new approach was needed.

The second approach called the sizeof operator in an unsafe context to determine the memory size (see figure 21). The sizeof operator is limited to unmanaged data types and can only be called in an unsafe context. An unsafe context is also known as an unmanaged context meaning memory needs to be managed manually *(Choudhary, 2015)*.



```
private unsafe void UnsafeSizeOf<T>() where T : unmanaged
{
    Debug.Log($"Size of {typeof(T)} is {sizeof(T)} bytes");
}
```

Figure 21 Experiment A.1 second approach (Kyaw, 2021)

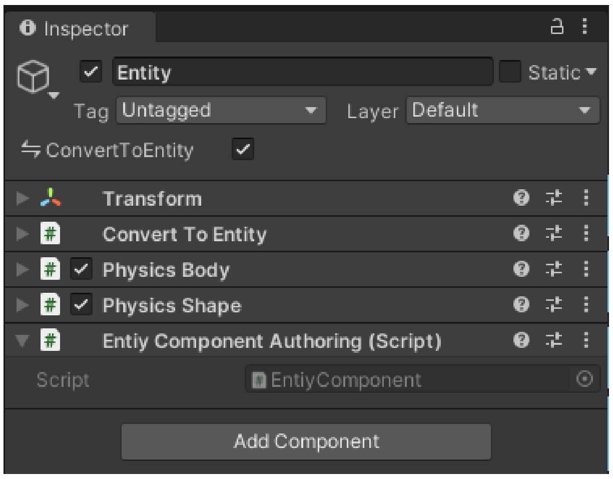
The resulting issue with this approach is only the memory of unmanaged data types can be determined. This results in types listed in the unity physics and container types not being analyzable using this approach.
The components that can't be analyzed within this experiment are moved to experiment A.2. The reason being that experiment A.2 analyses the memory allocation of monobehaviour which is a managed type allowing to analyze the memory of the managed components of unity physics and container types.

4.4 Implementation: Experiment A.2

This experiment analyzed the overhead when using monobehaviours and entities.
Through some complication in experiment A.1 the components listed in the table under unity physics and container types found in chapter 3.4.1 are analyzed within this experiment.
For this experiment three gameobjects were created for ECS components, classic Unity components and the container types. Each of these gameobjects were assigned with the corresponding components to be analyzed.

The ECS gameobject contained the following components as seen in figure 22. The EntityComponent used for the gameobject is an empty struct implementing the IComponentData interface.

The gameobjects for classic Unity contained the components corresponding to the ECS gameobject as seen in figure 23. The MonobehaviourClass used for this gameobject is an empty class inheriting the monobehaviour.

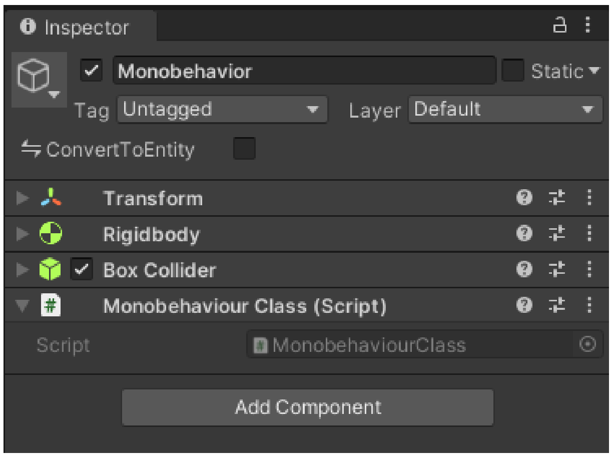

Inspector

✓ Entity Static
Tag Untagged Layer Default
ConvertToEntity ✓

Transform
Convert To Entity
✓ Physics Body
✓ Physics Shape
Entiy Component Authoring (Script)
Script EntiyComponent

Add Component

Figure 22 Experiment A.2 - Gameobject containing ECS components (Kyaw, 2022)


Inspector

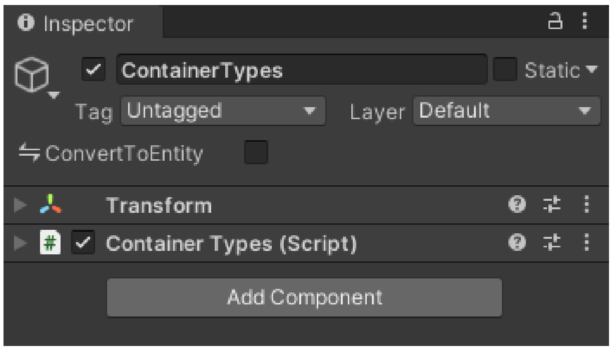
✓ Monobehavior Static
Tag Untagged Layer Default
ConvertToEntity

Transform
Rigidbody
✓ Box Collider
Monobehaviour Class (Script)
Script MonobehaviourClass

Add Component

Figure 23 Experiment A.2 - Gameobject containing classic Unity components (Kyaw, 2022)

The gameobject for the container types contained a script which created all container types to be tested with a length of 10 elements. (See figure 25)

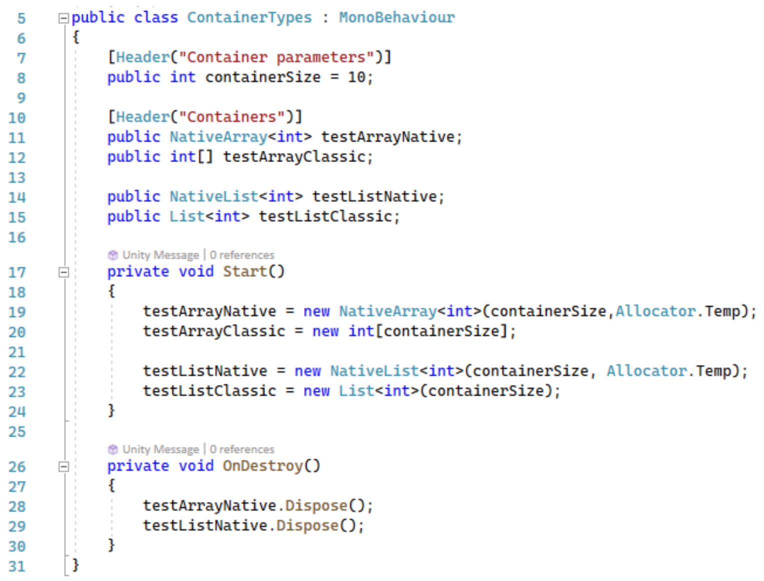

Inspector

✓ ContainerTypes Static
Tag Untagged Layer Default
ConvertToEntity

Transform
✓ Container Types (Script)

Add Component

Figure 24 Experiment A.2 - Gameobject for container types (Kyaw, 2022)



```
public class ContainerTypes : MonoBehaviour
{
    [Header("Container parameters")]
    public int containerSize = 10;

    [Header("Containers")]
    public NativeArray<int> testArrayNative;
    public int[] testArrayClassic;

    public NativeList<int> testListNative;
    public List<int> testListClassic;

    Unity Message | 0 references
    private void Start()
    {
        testArrayNative = new NativeArray<int>(containerSize,Allocator.Temp);
        testArrayClassic = new int[containerSize];

        testListNative = new NativeList<int>(containerSize, Allocator.Temp);
        testListClassic = new List<int>(containerSize);
    }

    Unity Message | 0 references
    private void OnDestroy()
    {
        testArrayNative.Dispose();
        testListNative.Dispose();
    }
}
```

Figure 25 Experiment A.2 - Container type instancing script (Kyaw, 2022)

The memory allocation size of the ECS components was measured using the new UI added by the 0.50 DOTS update. The new UI included a window that shows the currently present archetypes in the scene and a detailed list of all components associated to it. For the classic Unity types the memory profiler was used to analyze the memory by captured the memory data in a certain frame.

4.5 Experiment B.1

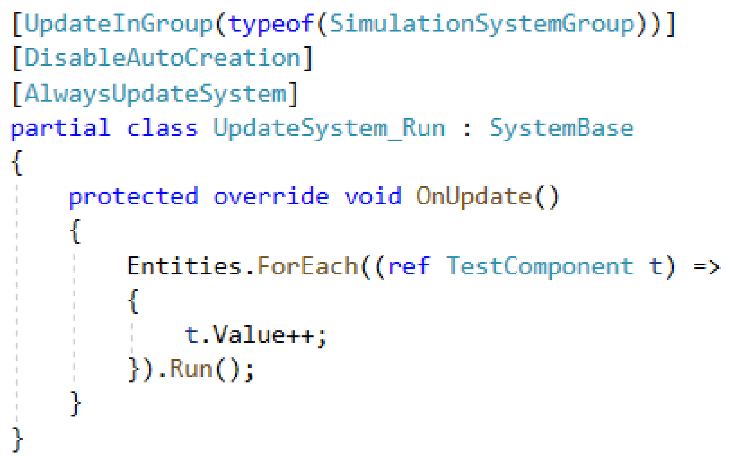
This experiment analyzed the read write speed of ECS code in comparison to classic Unity code.
For the implementation of this experiment the code samples provided by Unity Technologies were used as a base structure with sections being reused. The code samples used can be found on Unity Technologies official GitHub page under the following link:
(https://github.com/Unity-Technologies/EntityComponentSystemSamples/tree/master/ECSSamples/Assets/StressTests/ManySystems_Linear)

During the implementation process feature that were initially planned are changed. The experiment initially utilized a list containing ten objects for its simulation. Using only 10 objects resulted in nearly unmeasurably small results. The solution to this problem was implemented by adding a modifiable parameter to the number of simulation object in the list. Experiments will be conducted with a list of 100 objects.

The data structures AoS and SoA used to structure the objects were only implemented in classic Unity. The reason for this change was the vastly different way Unity DOTS allowed the creating of container types like lists. Unity DOTS required a dynamic buffer to be associated to an entity to represent a list or array in an object making the implementation considerably more complex. With the purpose of using the data structures being the analysis of the data-oriented principle called data locality the implementation in DOTS was not required and therefore cancelled.

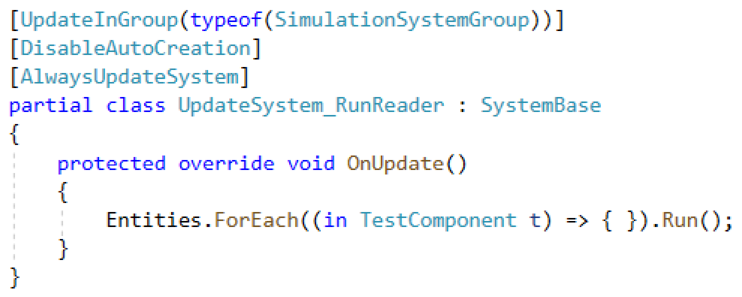
4.5.1 DOTS implementation

For the DOTS implementation of B.1, two systems were created which update all values within the list of simulation objects. The first system implemented a read and write which increments the float value of all entities containing the TestComponent component (see figure 26). The second system implemented the readonly system which only access the component (see figure 27). The implementations of the systems are as seen below.



```
[UpdateInGroup(typeof(SimulationSystemGroup))]
[DisableAutoCreation]
[AlwaysUpdateSystem]
partial class UpdateSystem_Run : SystemBase
{
    protected override void OnUpdate()
    {
        Entities.ForEach((ref TestComponent t) =>
        {
            t.Value++;
        }).Run();
    }
}
```

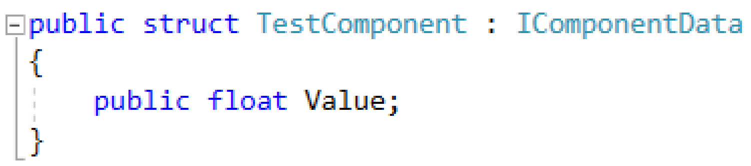
Figure 26 Experiment B.1-DOTS Read Write update system (Kyaw, 2021)



```
[UpdateInGroup(typeof(SimulationSystemGroup))]
[DisableAutoCreation]
[AlwaysUpdateSystem]
partial class UpdateSystem_RunReader : SystemBase
{
    protected override void OnUpdate()
    {
        Entities.ForEach((in TestComponent t) => { }).Run();
    }
}
```

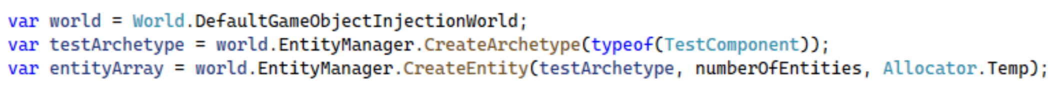
Figure 27 Experiment B.1-DOTS Read update system (Kyaw, 2021)

The entities for this experiment were instanced by creating an entity archetype containing the TestComponent and generating entities using that archetype. The figure below shows the implementation of the TestComponent and the entity creation function.



```
public struct TestComponent : IComponentData
{
    public float Value;
}
```

Figure 28 Experiment B.1-DOTS TestComponent (Kyaw, 2021)



```
var world = World.DefaultGameObjectInjectionWorld;
var testArchetype = world.EntityManager.CreateArchetype(typeof(TestComponent));
var entityArray = world.EntityManager.CreateEntity(testArchetype, numberOfEntities, Allocator.Temp);
```

Figure 29 Experiment B.1-DOTS entity creation function (Kyaw, 2021)

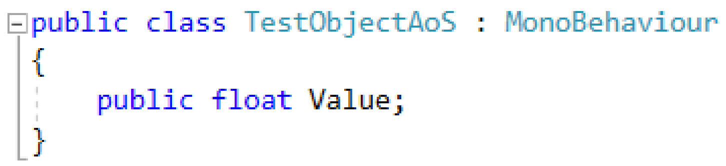
After every update call the time required for the system to update all entities is measured in milliseconds using the System.Diagnostics stopwatch and added to the stress test statistic. Every 1000 updates the average time to update all entities per update was outputted including a standard deviation. The experiment kept running until a set maximum number of update cycles was reached. The statistic used is described in 4.2 *Stress test assets*.

4.5.2 Classic Unity implementation

The implementation in Classic Unity is very similar to the DOTS implementation. In contrast to the feature implemented by the DOTS version, the classic Unity version

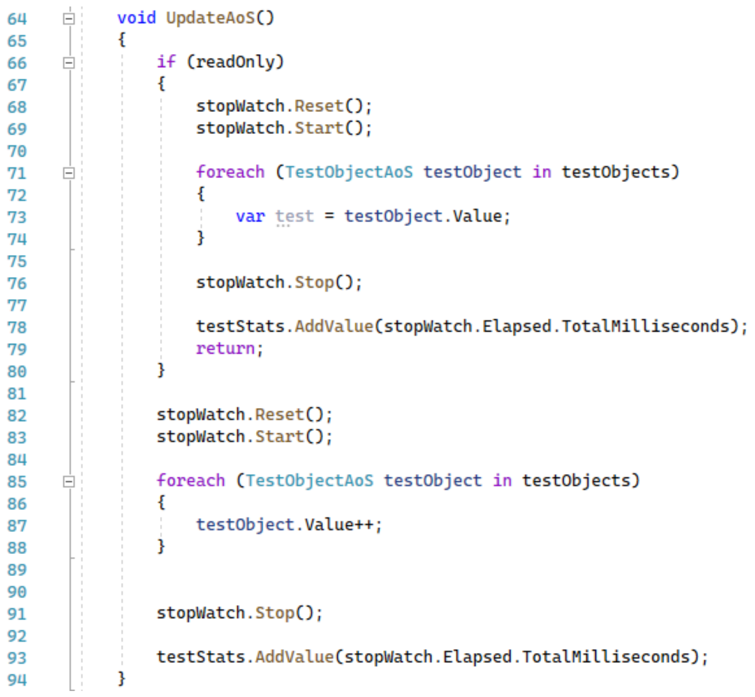
implemented the objects using the data structures AoS and SoA. As described in chapter 2.4.2 *data locality*, the AoS structure creates multiple objects containing floats while the SoA structure creates one object containing an array of floats.
Identical to the DOTS implementation, the update time was measured in milliseconds using the System.Diagnostics stopwatch and evaluated using the stress test statistic. The results of the statistics were outputted every 1000 updates until a set maximum.

The AoS implementation works identical to the DOTS implementation. A certain number of objects are created and updates by the update function.
The implementation of the function that updates the objects and the object itself can be found below. The read-write and readonly are implemented in the same function with readonly contained in the readonly if statement.



```
public class TestObjectAoS : MonoBehaviour
{
    public float Value;
}
```

Figure 30 Experiment B.1-Classic Unity AoS object (Kyaw, 2021)

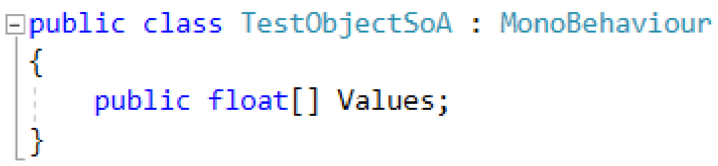


```
void UpdateAoS()
{
    if (readOnly)
    {
        stopWatch.Reset();
        stopWatch.Start();

        foreach (TestObjectAoS testObject in testObjects)
        {
            var test = testObject.Value;
        }

        stopWatch.Stop();

        testStats.AddValue(stopWatch.Elapsed.TotalMilliseconds);
        return;
    }

    stopWatch.Reset();
    stopWatch.Start();

    foreach (TestObjectAoS testObject in testObjects)
    {
        testObject.Value++;
    }


    stopWatch.Stop();

    testStats.AddValue(stopWatch.Elapsed.TotalMilliseconds);
}
```

Figure 31 Experiment B.1-Classic update system AoS (Kyaw, 2021)

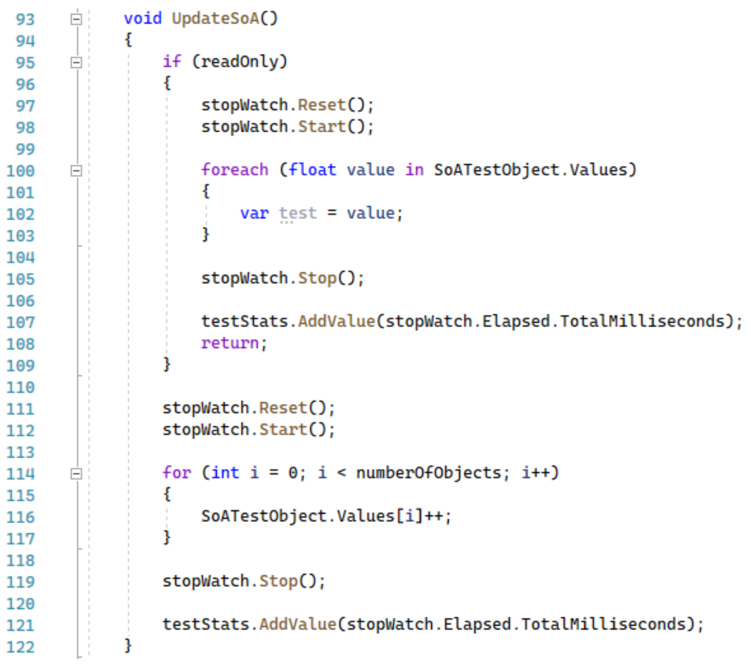
The SoA implementation in contrast did not create multiple objects. As previously mentioned, one object was created containing an array of the values with the array size representing the number of objects.

The implementation of the function that updates the array and the object itself can be found below. The read-write and readonly are implemented in the same function with readonly contained in the readonly if statement.



```
public class TestObjectSoA : MonoBehaviour
{
    public float[] Values;
}
```

Figure 32 Experiment B.1-Classic Unity SoA object



```
void UpdateSoA()
{
    if (readOnly)
    {
        stopWatch.Reset();
        stopWatch.Start();

        foreach (float value in SoATestObject.Values)
        {
            var test = value;
        }

        stopWatch.Stop();

        testStats.AddValue(stopWatch.Elapsed.TotalMilliseconds);
        return;
    }

    stopWatch.Reset();
    stopWatch.Start();

    for (int i = 0; i < numberOfObjects; i++)
    {
        SoATestObject.Values[i]++;
    }

    stopWatch.Stop();

    testStats.AddValue(stopWatch.Elapsed.TotalMilliseconds);
}
```

Figure 33 Experiment B.1-Classic update system SoA (Kyaw, 2021)

4.6 Experiment B.2

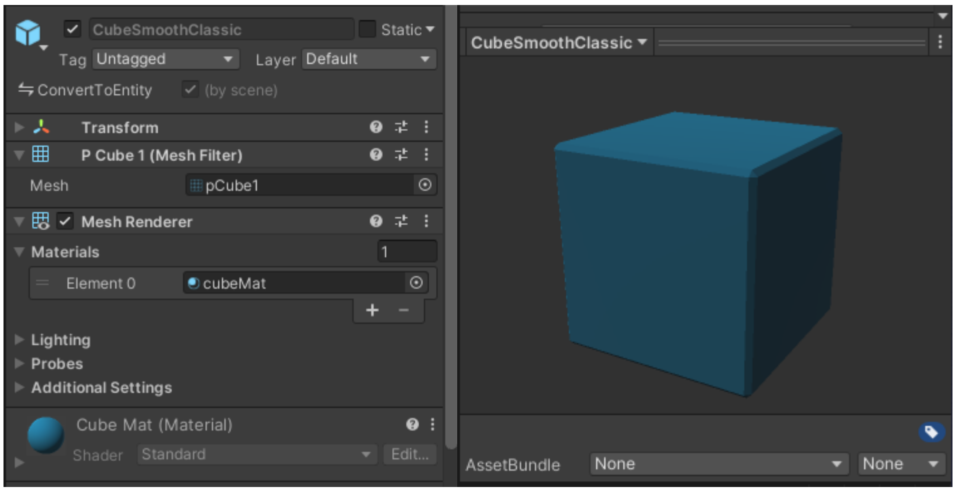
This experiment conducted load test on the creation of gameobject and entities including entity conversion.
This experiment was not implemented and conducted as a standalone experiment as initially planned. Instead, the load testing was conducted in combination with experiment C. Experiment C analyzed the render performance of Unity DOTS comparatively to classic Unity by creating large amounts of objects and entities. This made experiment C ideal for performing the load-testing of experiment B.2.

4.7 Implementation: Experiment C

This experiment analyzed the capability of the DOTS renderer in comparison to the renderer of classic Unity. In addition, due to some changes in experiment B.2, this experiment also conducted load tests on the creation of game objects and entities including the entity conversion.
This experiment instanced objects/entities using three different methods

The method for classic Unity utilized prefabs with the minimum requirements of components to render a cube. The components include a MeshFilter, MeshRenderer and a material. (See figure 34)
The method for Unity DOTS created entities using the pure ECS method by creating an entity archetype with the needed components to render a cube. The components used for the entity can be seen in figure 35.

The last method was the hybrid version between gameobjects and entity, entity conversion. For the entity conversion the gameobject from the classic Unity method was used. In order to account for a worst-case scenario, the experiment contained a setting that allowed the gameobject to be converted on creation instead of instancing the entity from the pre-compiled entity. (See figure 36)


CubeSmoothClassic   Static
Tag Untagged   Layer Default
ConvertToEntity   ✓ (by scene)
Transform
P Cube 1 (Mesh Filter)
Mesh   pCube1
✓ Mesh Renderer
Materials   1
Element 0   cubeMat
Lighting
Probes
Additional Settings
Cube Mat (Material)
Shader   Standard   Edit...

CubeSmoothClassic
AssetBundle   None   None

Figure 34 Experiment C Gameobject cube (Kyaw, 2021)

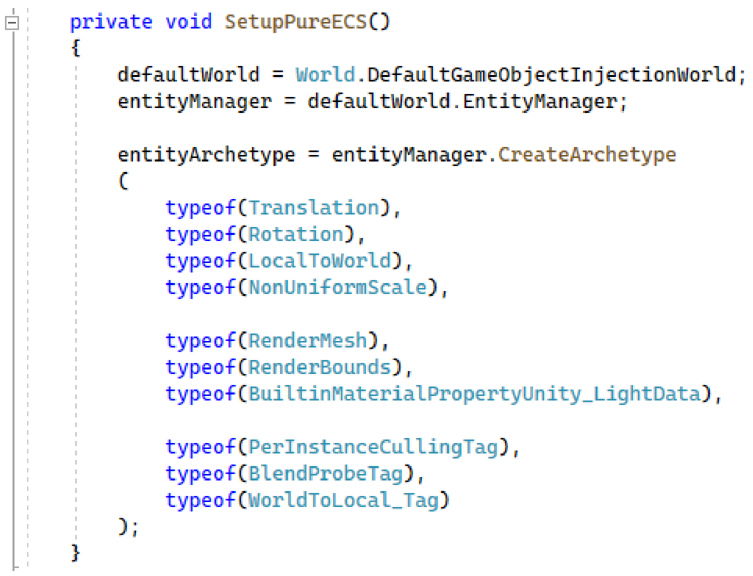

```
private void SetupPureECS()
{
    defaultWorld = World.DefaultGameObjectInjectionWorld;
    entityManager = defaultWorld.EntityManager;

    entityArchetype = entityManager.CreateArchetype
    (
        typeof(Translation),
        typeof(Rotation),
        typeof(LocalToWorld),
        typeof(NonUniformScale),

        typeof(RenderMesh),
        typeof(RenderBounds),
        typeof(BuiltinMaterialPropertyUnity_LightData),

        typeof(PerInstanceCullingTag),
        typeof(BlendProbeTag),
        typeof(WorldToLocal_Tag)
    );
}
```

Figure 35 Experiment C Entity cube (Kyaw, 2021)

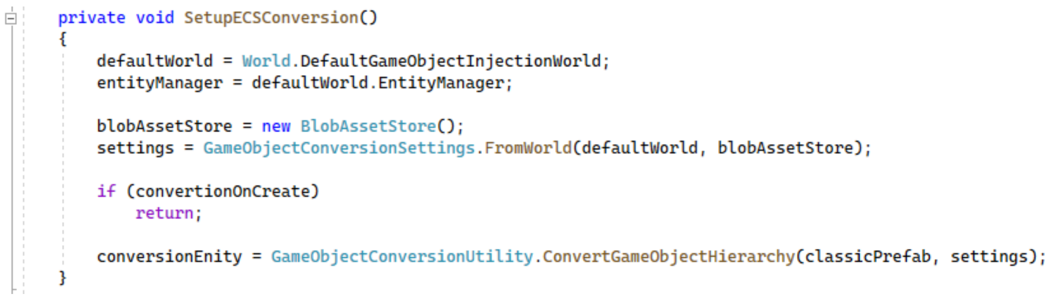
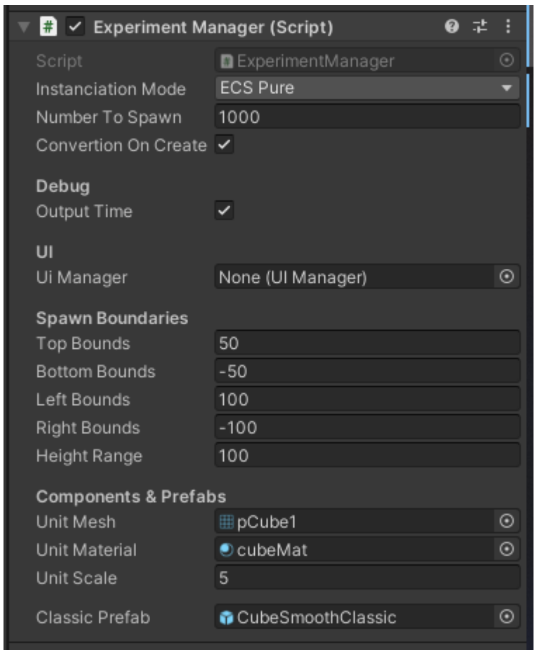

```
private void SetupECSConversion()
{
    defaultWorld = World.DefaultGameObjectInjectionWorld;
    entityManager = defaultWorld.EntityManager;

    blobAssetStore = new BlobAssetStore();
    settings = GameObjectConversionSettings.FromWorld(defaultWorld, blobAssetStore);

    if (convertionOnCreate)
        return;

    conversionEnity = GameObjectConversionUtility.ConvertGameObjectHierarchy(classicPrefab, settings);
}
```

Figure 36 Experiment C Entity Conversion of gameobject cube (Kyaw, 2021)

Before the experiment the parameters for the experiment were set in the inspector interface (see figure 37) of the Experiment Manager depending on the subject to analyze. The top three parameters modify the experiment. All other parameters were not changed from the values see in figure 37.
During the experiment the space bar button was pressed to instance a certain number of objects/entities. The number of objects/entities and method of instancing them were determined by the parameters in the experiment manager. All instanced objects/entities were assigned a random scale and a random position within the view frustrum of the camera determined by the spawn boundaries.

Throughout the experiment the current number of instanced objects, the mode at which the objects were created and the current fps ("frames per second") were displayed on the game screen as seen in figure 38. The results of experiment C were gathered in max polygon count to render in 60 fps.

For the purpose of analyzing the subject of experiment B.2 the time to instance the objects/entities per instance call (pressing space bar) was measured using the stopwatch as described in experiment B.1. The results for experiment B.2 were gathered in milliseconds per instance call.


Experiment Manager (Script)

Script	ExperimentManager
Instanciation Mode	ECS Pure
Number To Spawn	1000
Convertion On Create	✓
Debug	
Output Time	✓
UI	
Ui Manager	None (UI Manager)
Spawn Boundaries	
Top Bounds	50
Bottom Bounds	-50
Left Bounds	100
Right Bounds	-100
Height Range	100
Components & Prefabs	
Unit Mesh	pCube1
Unit Material	cubeMat
Unit Scale	5
Classic Prefab	CubeSmoothClassic

Figure 37 Experiment C experiment manager inspector interface (Kyaw, 2021)

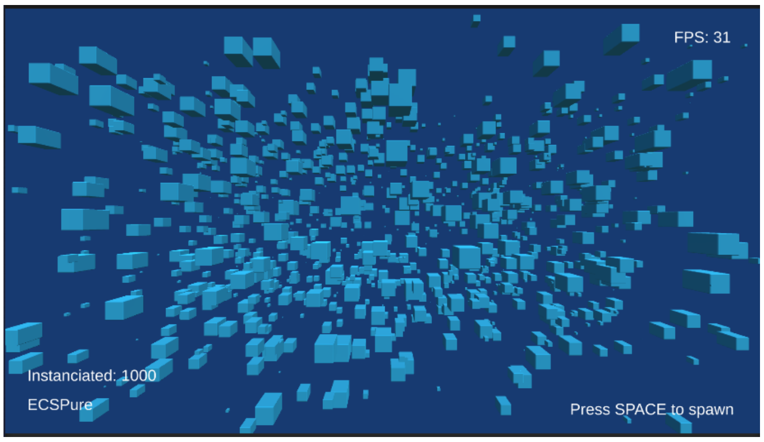


Figure 38 Experiment C game window (Kyaw, 2021)

4.8 Experiment D

This experiment analyzed the multithreading solution the job system in comparison to classic Unity including basic C# threading. To analyze this subject the experiment implemented a merge sort algorithm in each of the different environment. The implementation of the merge sort algorithm in DOTS was drastically changed. The merge sort algorithm was not implemented in ECS with multithreading and ECS without multithreading. Instead, the multithreaded merge sort was implemented in classic unity using the job system because of the problem described in experiment B.1 with arrays in ECS components.
With all versions implemented in classic Unity a more isolated evaluation of the job system's performance was made in comparison to classic unity and C# threading.

The experiment was conducted in classic Unity without multithreading, with basic C# threading and with the job system. The System.Diagnostics stopwatch measured the elapsed time to sort the array and outputted the time in millisecond and nanoseconds.
The algorithm implemented in this experiment is a basic recursive version of merge sort. The algorithm sorts an array of numbers by splitting up the array repeatedly into smaller sections to sort until only one number is left. These sections are merged back together in an ascending order until the array becomes whole. A visual representation of how the algorithm works can be seen in figure 39.

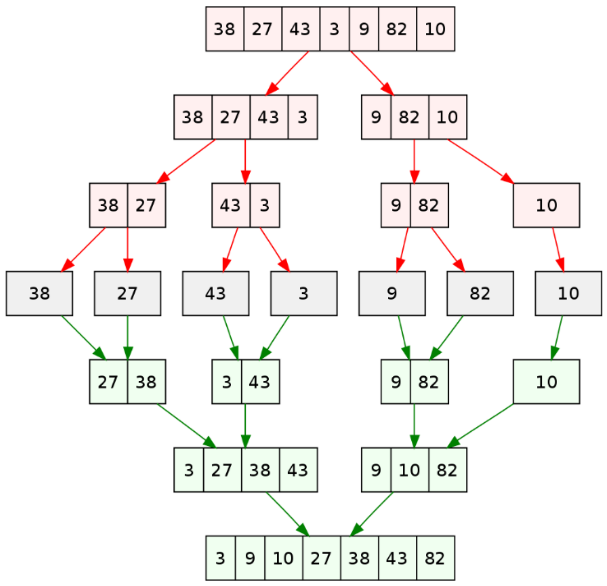
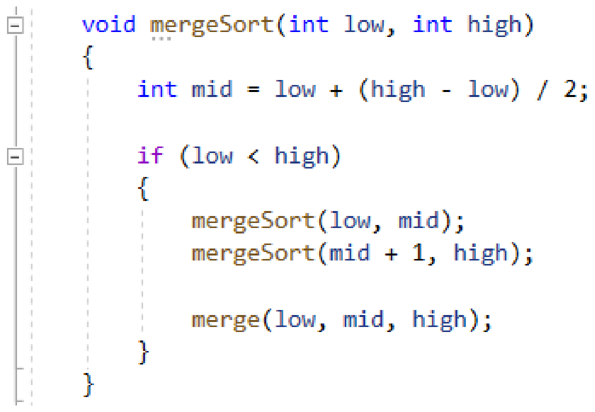


Figure 39 Visual representation of a merge sort algorithm (Baumann, no date)

4.8.1 Classic Unity implementation

The classic Unity implementation created an array containing random integer values. That array was sorted using the recursive merge sort algorithm described previously in chapter 4.8 *Experiment D*.

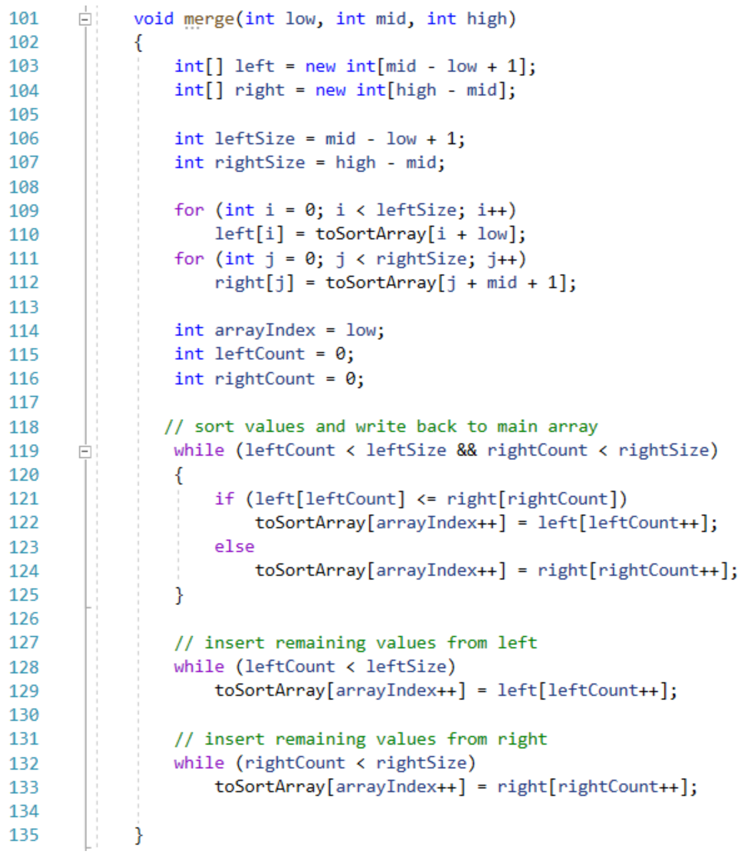
The function that split up the array into smaller section using recursion can be seen in figure 40. The low and high parameter determine the index range of the array that is currently analyzed.



```
void mergeSort(int low, int high)
{
    int mid = low + (high - low) / 2;

    if (low < high)
    {
        mergeSort(low, mid);
        mergeSort(mid + 1, high);

        merge(low, mid, high);
    }
}
```

Figure 40 Experiment D recursive merge sort call (Kyaw, 2021)

The process of merging the section back together in an ascending order was implemented by the merge function. The merge function compared the values of two sections which are determined by the parameters and rearranges them in an ascending order. The two sections range from low to mid and from mid to high. (See figure 41)



```
void merge(int low, int mid, int high)
{
    int[] left = new int[mid - low + 1];
    int[] right = new int[high - mid];

    int leftSize = mid - low + 1;
    int rightSize = high - mid;

    for (int i = 0; i < leftSize; i++)
        left[i] = toSortArray[i + low];
    for (int j = 0; j < rightSize; j++)
        right[j] = toSortArray[j + mid + 1];

    int arrayIndex = low;
    int leftCount = 0;
    int rightCount = 0;

   // sort values and write back to main array
    while (leftCount < leftSize && rightCount < rightSize)
    {
        if (left[leftCount] <= right[rightCount])
            toSortArray[arrayIndex++] = left[leftCount++];
        else
            toSortArray[arrayIndex++] = right[rightCount++];
    }

    // insert remaining values from left
    while (leftCount < leftSize)
        toSortArray[arrayIndex++] = left[leftCount++];

    // insert remaining values from right
    while (rightCount < rightSize)
        toSortArray[arrayIndex++] = right[rightCount++];

}
```

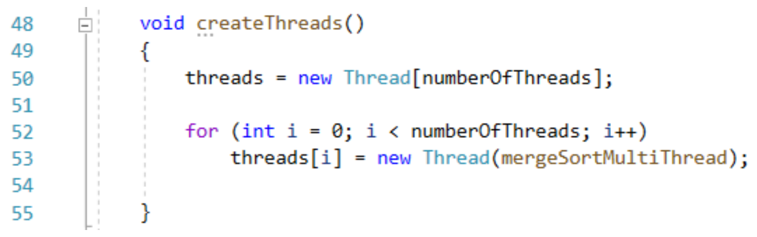
Figure 41 Experiment D merge sort algorithm behavior (Kyaw, 2021)

4.8.2 Classic Unity multithreaded implementation

The multithreaded version of classic unity implemented basic C# threading using the System.Threading threads. This version utilizes the same algorithm as the none-multithreaded version with some extended functionality for multithreading.

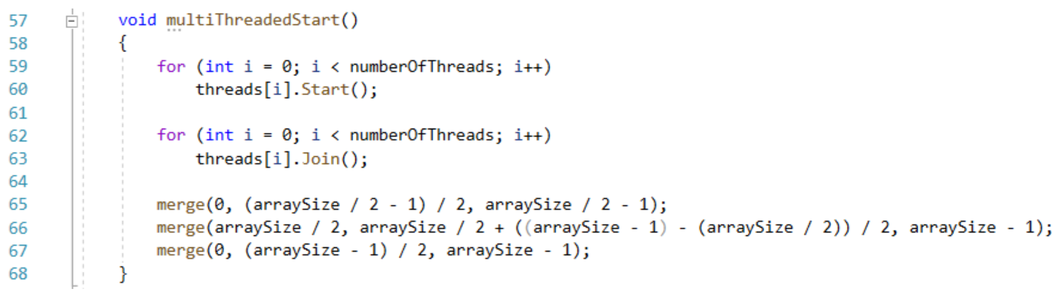
The createThreads function instanced the needed threads assigned with the method to be executed on them (see figure 42). When the threads are started, the implementation split up the main array into four sections by assigning each thread a range of the array depending on the index of the thread in an array. Each thread executed the merge sort on its own section. (See figure 44)

After the threads are finished the sections were merged back together. To ensure that the four section were merged together correctly the join function is called on all threads. By calling the join function on all threads the main thread waited for all thread to finish executing to resume. (See figure 43)



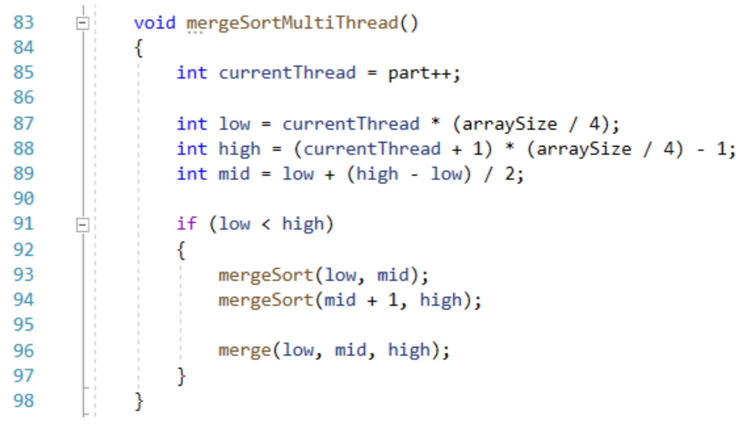
```
void createThreads()
{
    threads = new Thread[numberOfThreads];

    for (int i = 0; i < numberOfThreads; i++)
        threads[i] = new Thread(mergeSortMultiThread);

}
```

Figure 42 Experiment D creation of threads (Kyaw, 2021)



```
void multiThreadedStart()
{
    for (int i = 0; i < numberOfThreads; i++)
        threads[i].Start();

    for (int i = 0; i < numberOfThreads; i++)
        threads[i].Join();

    merge(0, (arraySize / 2 - 1) / 2, arraySize / 2 - 1);
    merge(arraySize / 2, arraySize / 2 + ((arraySize - 1) - (arraySize / 2)) / 2, arraySize - 1);
    merge(0, (arraySize - 1) / 2, arraySize - 1);
}
```

Figure 43 Experiment D initiation function for multithreaded version (Kyaw, 2021)



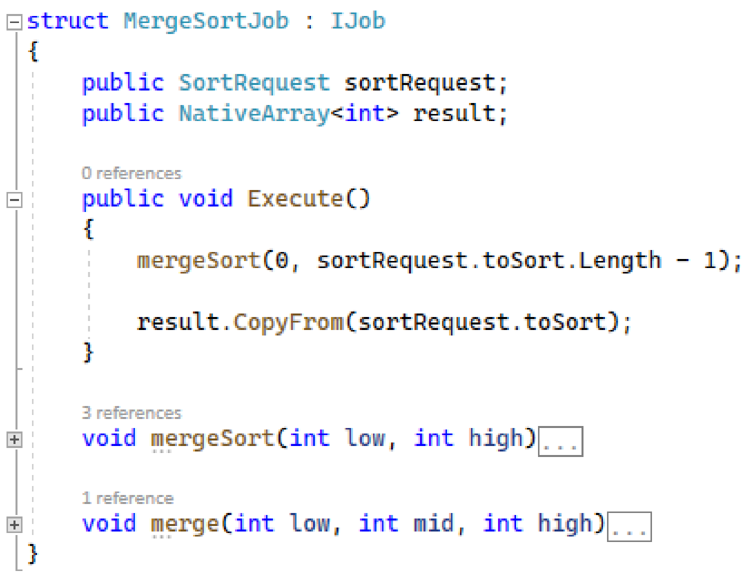
```
void mergeSortMultiThread()
{
    int currentThread = part++;

    int low = currentThread * (arraySize / 4);
    int high = (currentThread + 1) * (arraySize / 4) - 1;
    int mid = low + (high - low) / 2;

    if (low < high)
    {
        mergeSort(low, mid);
        mergeSort(mid + 1, high);

        merge(low, mid, high);
    }
}
```

Figure 44 Experiment D multithreaded version of the mergeSort function (Kyaw, 2021)

4.8.3 Job system implementation

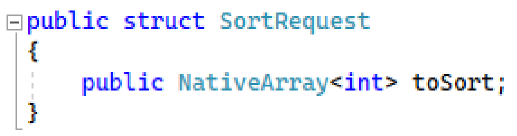
The multithreaded Job system implementation followed a similar structure as the multithreaded classic implementation. The Job system implemented the merge sort algorithm within a job assigned with a SortRequest which contains a copy of the section of the array to sort (see figure 45).

Each job was then created assigned to a section of the array (see figure 47). To recombine the section after each job had finished merge jobs were created which contained the merge function of the merge sort algorithm (see figure 48). The MergeRequest for the MergeJob jobs contained results of two MergeSortJob jobs.



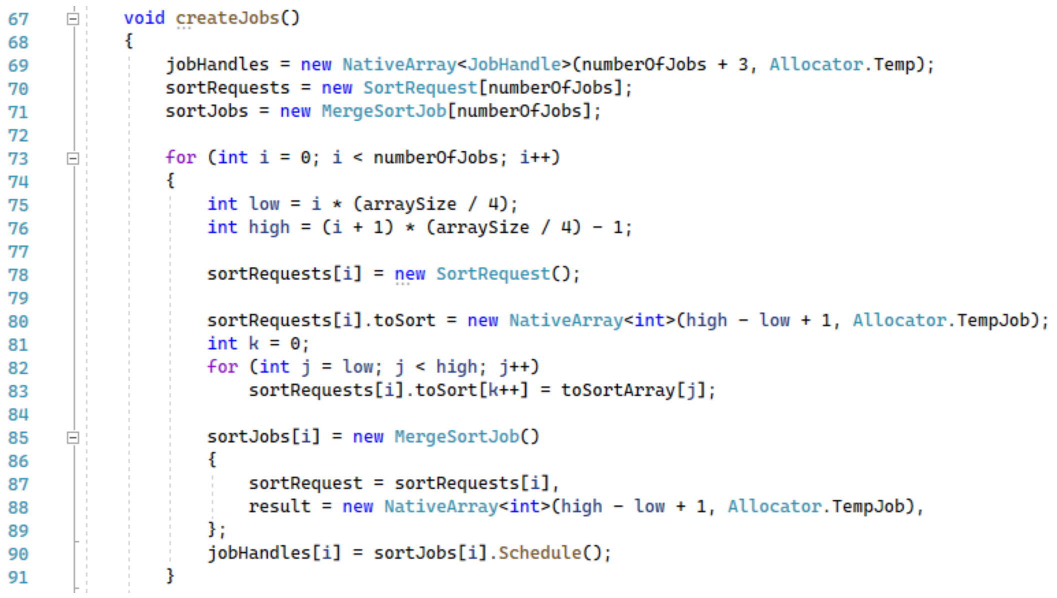
```
struct MergeSortJob : IJob
{
    public SortRequest sortRequest;
    public NativeArray<int> result;

    0 references
    public void Execute()
    {
        mergeSort(0, sortRequest.toSort.Length - 1);

        result.CopyFrom(sortRequest.toSort);
    }

    3 references
    void mergeSort(int low, int high)...

    1 reference
    void merge(int low, int mid, int high)...
}
```

Figure 45 Experiment D merge sort job (Kyaw, 2021)



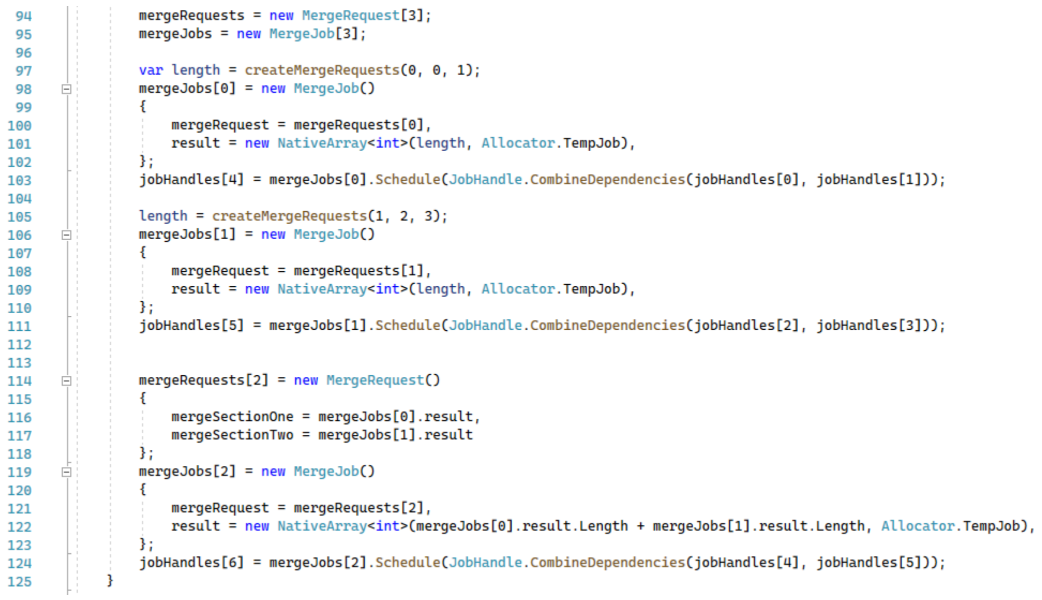
```
public struct SortRequest
{
    public NativeArray<int> toSort;
}
```

Figure 46 Experiment D SortRequest for MergeSortJob (Kyaw, 2022)



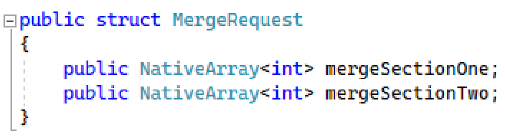
```
void createJobs()
{
    jobHandles = new NativeArray<JobHandle>(numberOfJobs + 3, Allocator.Temp);
    sortRequests = new SortRequest[numberOfJobs];
    sortJobs = new MergeSortJob[numberOfJobs];

    for (int i = 0; i < numberOfJobs; i++)
    {
        int low = i * (arraySize / 4);
        int high = (i + 1) * (arraySize / 4) - 1;

        sortRequests[i] = new SortRequest();

        sortRequests[i].toSort = new NativeArray<int>(high - low + 1, Allocator.TempJob);
        int k = 0;
        for (int j = low; j < high; j++)
            sortRequests[i].toSort[k++] = toSortArray[j];

        sortJobs[i] = new MergeSortJob()
        {
            sortRequest = sortRequests[i],
            result = new NativeArray<int>(high - low + 1, Allocator.TempJob),
        };
        jobHandles[i] = sortJobs[i].Schedule();
    }
```

Figure 47 Experiment D createJobs part 1: creating merge sort jobs (Kyaw 2021)



```
    mergeRequests = new MergeRequest[3];
    mergeJobs = new MergeJob[3];

    var length = createMergeRequests(0, 0, 1);
    mergeJobs[0] = new MergeJob()
    {
        mergeRequest = mergeRequests[0],
        result = new NativeArray<int>(length, Allocator.TempJob),
    };
    jobHandles[4] = mergeJobs[0].Schedule(JobHandle.CombineDependencies(jobHandles[0], jobHandles[1]));

    length = createMergeRequests(1, 2, 3);
    mergeJobs[1] = new MergeJob()
    {
        mergeRequest = mergeRequests[1],
        result = new NativeArray<int>(length, Allocator.TempJob),
    };
    jobHandles[5] = mergeJobs[1].Schedule(JobHandle.CombineDependencies(jobHandles[2], jobHandles[3]));


    mergeRequests[2] = new MergeRequest()
    {
        mergeSectionOne = mergeJobs[0].result,
        mergeSectionTwo = mergeJobs[1].result
    };
    mergeJobs[2] = new MergeJob()
    {
        mergeRequest = mergeRequests[2],
        result = new NativeArray<int>(mergeJobs[0].result.Length + mergeJobs[1].result.Length, Allocator.TempJob),
    };
    jobHandles[6] = mergeJobs[2].Schedule(JobHandle.CombineDependencies(jobHandles[4], jobHandles[5]));
}
```

Figure 48 Experiment D createJobs part 2: creating merge jobs (Kyaw, 2021)



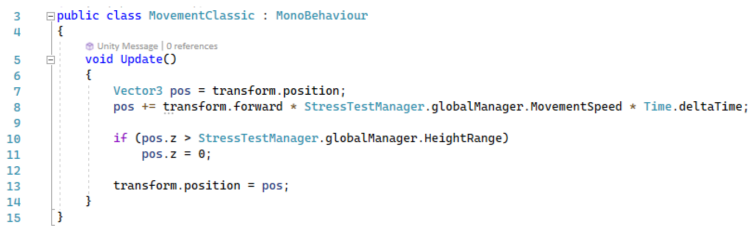
```
public struct MergeRequest
{
    public NativeArray<int> mergeSectionOne;
    public NativeArray<int> mergeSectionTwo;
}
```

Figure 49 Experiment D MergeRequest for MergeJob (Kyaw, 2022)

4.9 Implementation: Experiment E

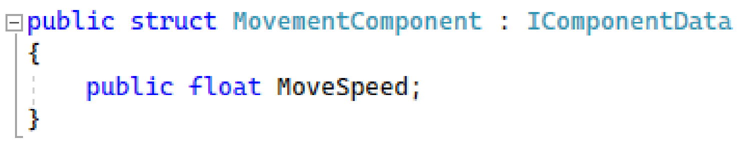
This experiment implemented a stress test that tests all of the various aspects analyzed in the previous experiments. The render performance was tested using the systems implemented in experiment C. In addition, the performance of the code was tested by implementing a movement system for both environments, classic Unity and ECS. To account for multithreading of the job system, the movement system in ECS was additionally implemented with the job system. In order to test the effectiveness of the Burst compiler to multithreaded code, the multithreaded ECS code allowed for the activation of Burst.

This experiment used experiment C as the base and added movement systems for each implementation environment. The movement systems implementation was nearly identical to each other with minimal changes to account for multithreading and ECS. The classic Unity implementation added monobehaviour to each object that moved itself. The ECS as defined used a system that moved all entities containing a Translation and MovementComponent component. The cubes that were spawned by the experiment C were moved away from the camera until a set maximum distance was reached. After the distance was reached the cubes were placed back close to the camera.
The implementation of the systems is as seen in the following figures.



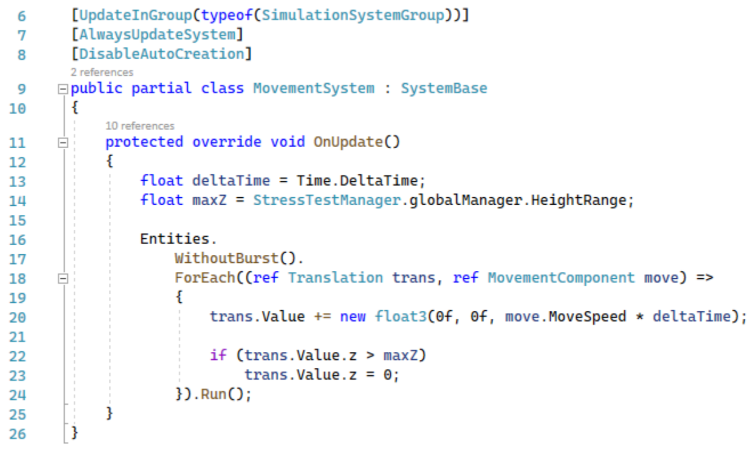
```
public class MovementClassic : MonoBehaviour
{
    Unity Message | 0 references
    void Update()
    {
        Vector3 pos = transform.position;
        pos += transform.forward * StressTestManager.globalManager.MovementSpeed * Time.deltaTime;

        if (pos.z > StressTestManager.globalManager.HeightRange)
            pos.z = 0;

        transform.position = pos;
    }
}
```

Figure 50 Experiment E movement system classic Unity (Kyaw, 2022)



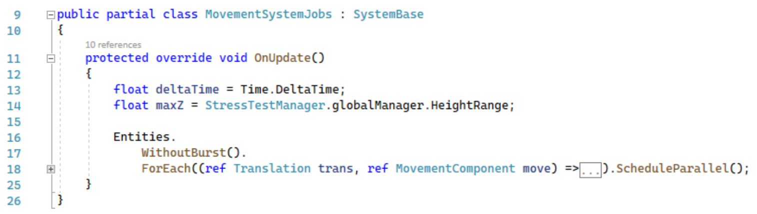
```
public struct MovementComponent : IComponentData
{
    public float MoveSpeed;
}
```

Figure 51 Experiment E MovementComponent for ECS systems (Kyaw, 2022)



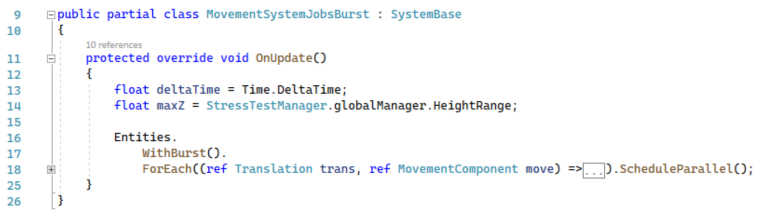
```
[UpdateInGroup(typeof(SimulationSystemGroup))]
[AlwaysUpdateSystem]
[DisableAutoCreation]
2 references
public partial class MovementSystem : SystemBase
{
    10 references
    protected override void OnUpdate()
    {
        float deltaTime = Time.DeltaTime;
        float maxZ = StressTestManager.globalManager.HeightRange;

        Entities.
            WithoutBurst().
            ForEach((ref Translation trans, ref MovementComponent move) =>
            {
                trans.Value += new float3(0f, 0f, move.MoveSpeed * deltaTime);

                if (trans.Value.z > maxZ)
                    trans.Value.z = 0;
            }).Run();
    }
}
```

Figure 52 Experiment E movement system ECS (Kyaw, 2022)



```
public partial class MovementSystemJobs : SystemBase
{
    10 references
    protected override void OnUpdate()
    {
        float deltaTime = Time.DeltaTime;
        float maxZ = StressTestManager.globalManager.HeightRange;

        Entities.
            WithoutBurst().
            ForEach((ref Translation trans, ref MovementComponent move) =>...).ScheduleParallel();
    }
}
```

Figure 53 Experiment E movement system ECS multithreaded with Job system (Kyaw, 2022)



```
public partial class MovementSystemJobsBurst : SystemBase
{
    10 references
    protected override void OnUpdate()
    {
        float deltaTime = Time.DeltaTime;
        float maxZ = StressTestManager.globalManager.HeightRange;

        Entities.
            WithBurst().
            ForEach((ref Translation trans, ref MovementComponent move) =>...).ScheduleParallel();
    }
}
```

Figure 54 Experiment E movement system ECS multithreaded enhanced with Burst (Kyaw, 2022)

The results for the render performance were gathered like in experiment C by analyzing the max polygon count to render in 60 fps. Because this experiment used experiment C as a base the load testing for the instantiation of objects and entities is included and measured in in milliseconds per instantiation.
The code performance was measured by analyzing the average time required for the movement systems to execute per frame depending on the number of objects/entities using the Unity profiler.

5 Conclusion

5.1 Evaluation

During this chapter the results of each experiment were presented and evaluated for each experiment individually in order to give a verdict to each research objective of the experiments.

5.1.1 Hardware - Testing Environment

As stated in 3.3.2 *Hardware Benchmark*, the hardware was benchmarked. The testing hardware was not changed from the device described in 3.3.1 *Hardware Specifications*. The benchmarks performed were limited to benchmarking with the UserBenchmark because it offers a detailed evaluation of all hardware components and comparison to other benchmarks of the same hardware. The results of the benchmark can be found under the following link.
https://www.userbenchmark.com/UserRun/54630844

5.1.2 Evaluation: Experiment A.1

This experiment analyzed the memory allocation size difference of data types in classic Unity in comparison to Unity DOTS. The following diagram presents the results of the experiment in the following data table in bytes. The results showed that the classic Unity data types allocated the same amount of memory as their Unity DOTS corresponding data types. The data types analyzed were all unmanaged data types.
This experiment evaluated that unmanaged data types in Unity and Unity DOTS allocated the same amount of memory.

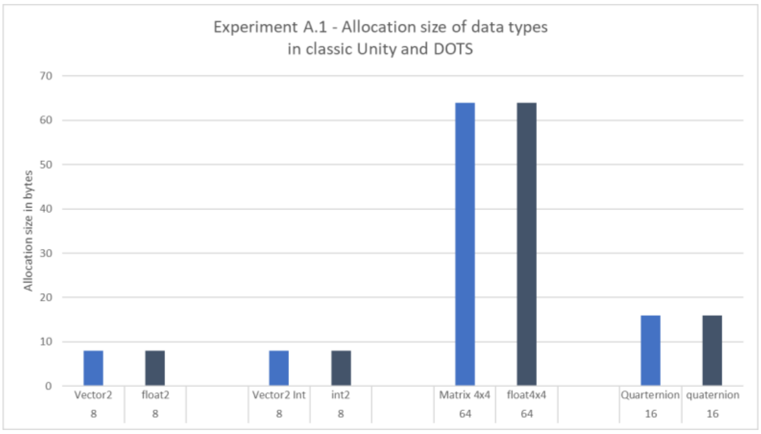
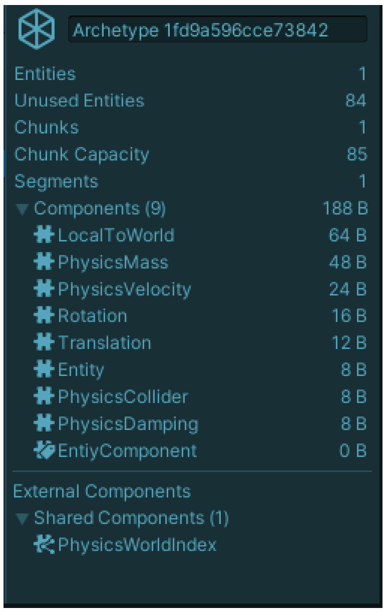


Figure 55 Results of experiment A.1 (Kyaw, 2022)

5.1.3 Evaluation: Experiment A.2

This experiment analyzed the memory overhead when inheriting from monobehaviour and of ECS components. The physics components and container types of experiment A.1 were analyzed within this experiment. The results are presented in a table as a comparison between classic Unity and Unity DOTS.

The memory size results of the physics components were combined because the DOTS components for physics were hard to determine. The initially components Physics Body and Physics Shape compiled into Physics Mass, Velocity, Collider and Damping as seen in figure 56.



Archetype 1fd9a596cce73842

Entities	1
Unused Entities	84
Chunks	1
Chunk Capacity	85
Segments	1
Components (9)	188 B
LocalToWorld	64 B
PhysicsMass	48 B
PhysicsVelocity	24 B
Rotation	16 B
Translation	12 B
Entity	8 B
PhysicsCollider	8 B
PhysicsDamping	8 B
EntiyComponent	0 B

External Components
Shared Components (1)
PhysicsWorldIndex

Figure 56 Results of Experiment A.2 - ECS archetype (Kyaw, 2022)

Classic Unity	Unity DOTS
Overhead	
Monobehaviour 488 Bytes	IComponentData 0 Bytes
Unity Physics	
Rigidbody & Box Collider 304 + 328 Bytes = 632 Bytes	Physics Body & Physics Shape 88 Bytes
Containers types	
Array [] 8 Bytes	Native Array 56 Bytes
List<N> 8 Bytes	Native List 48 Bytes

Table 2 Results of Experiment A.2 - memory comparison of managed types (Kyaw, 2022)

The result showed that monobehaviour inheriting script had an inherit 488 bytes of memory allocation before any field were declared. In comparison the struct that holds the data in ECS had no initial memory allocation because it implemented an interface with no required fields.
The results for the physics components showed that classic Unity required around 632 bytes of components for its physics simulation while the ECS only required around 88 bytes of data.
All container types for this experiment were initialized with a length of 10 elements. The container types which are usually used for the job system allocated 56 bytes for a native array and 48 bytes for a native list. The classic container in comparison allocated both 8 bytes.

This experiment evaluated that Unity DOTS required less memory on average for the use of engine components such as the physics simulation.

5.1.4 Evaluation: Experiment B.1

This experiment analyzed the memory access speed of ECS code in comparison to classic Unity code. The results of this experiment are presented as graphs. Each graph in the diagrams represent a dataset of approximately 10.000 runs with the x axis describing the number of runs executed and the y axis describing the time elapsed per frame to update 100 objects in milliseconds.

The first section of the experiment was the classic Unity AoS system whose read-write system required an average of ~0.0022ms +- ~0.0002ms to update all objects per frame. The readonly AoS system required an average of ~0.0015ms +- ~0.00064ms per frame. (See figure 57)

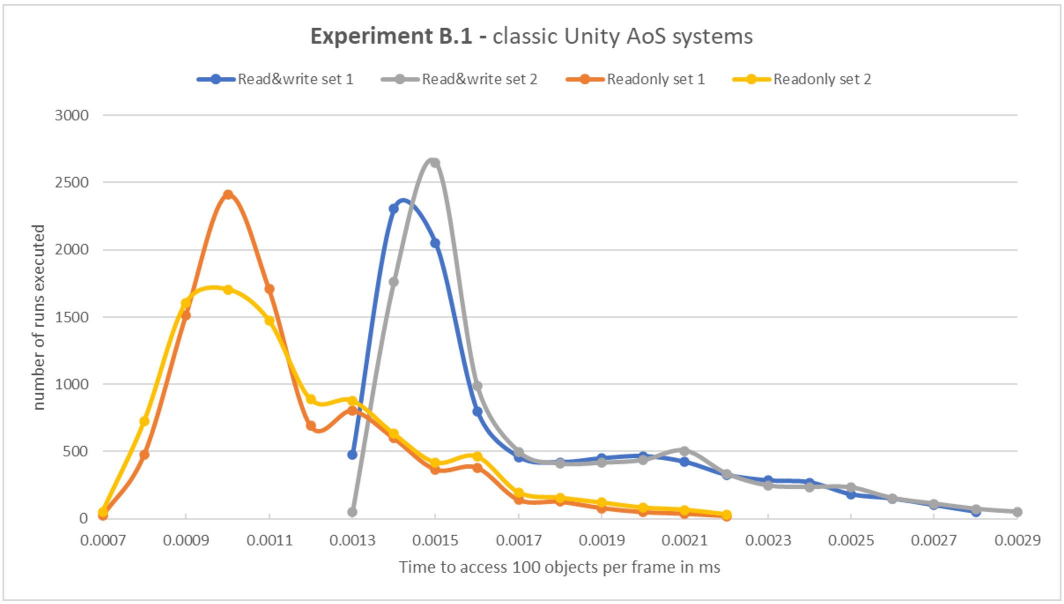


Figure 57 Results of experiment B.1 – Classic Unity AoS systems (Kyaw, 2022)

The other classic Unity system was the SoA system whose read-write system required an average of ~0.00074ms +- ~0.0004ms per frame. While the readonly system required an average of ~0.001ms +- ~0.0006ms per frame. The fact that the readonly system required more time on average to update all objects indicated that accessing the data of an array without modifying it seems more taxing to the system. (See figure 58)

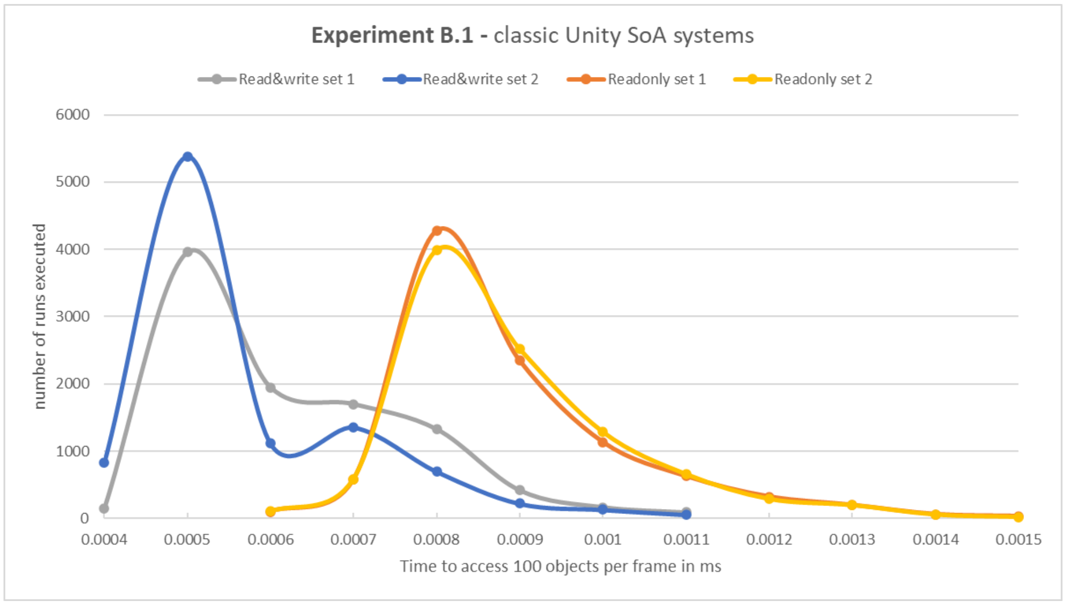


Figure 58 Results of experiment B.1 – Classic Unity SoA systems (Kyaw, 2022)

One of the objectives of this experiment was the analysis of the performance impact through the use of the data structure AoS and SoA from data locality. The difference in performance became apparent when comparing the two data structure implementations.

In the read-write system the SoA implementation's average worst performance was still better than AoS best performance. The SoA read-write performed around ~200% faster on average than the AoS implementation. (See figure 59)
The outcome of the performance comparison on the readonly systems was different compared to the read-write comparison. The SoA readonly system performed around ~33% slower than the AoS system. (See figure 60)

This concluded that considering data locality has a huge impact on code performance even with simple changes such as the structure of the data. Additional iterating through an array without modifying it proved to be slower than actually modifying it.

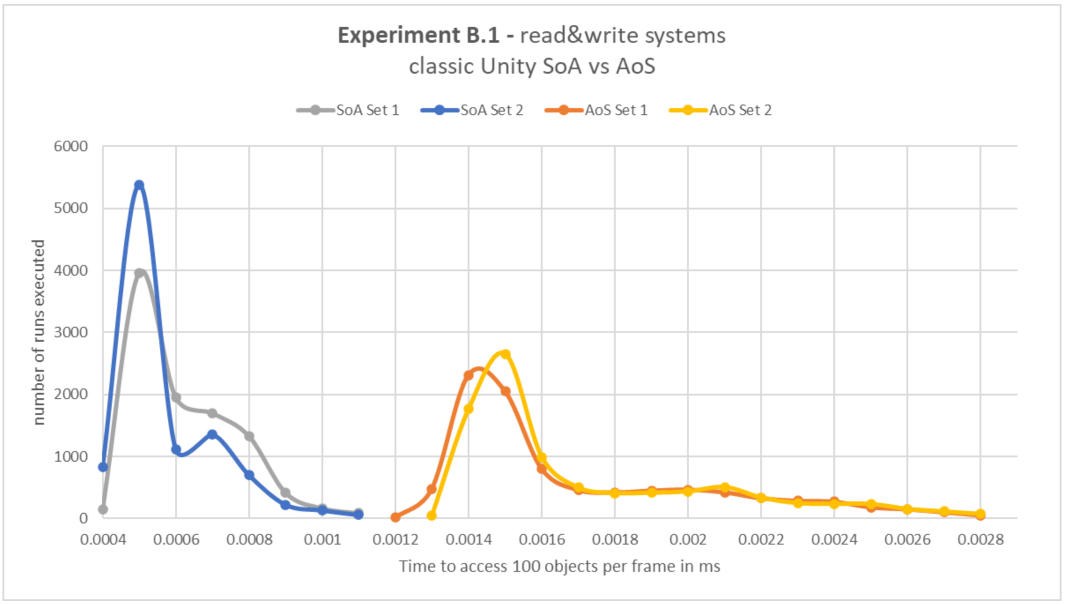


Figure 59 Results of experiment B.1 - comparison of read-write systems in classic Unity (Kyaw, 2022)

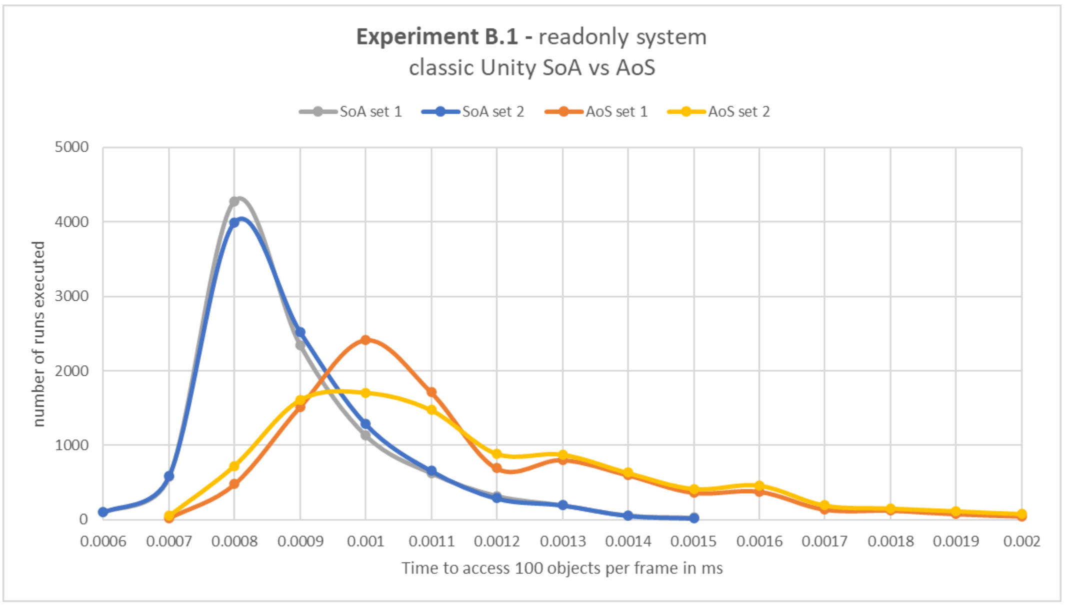


Figure 60 Results of experiment B.1 - comparison of readonly systems in classic Unity (Kyaw, 2022)

On the contrary to the classic Unity systems there were the ECS systems whose read-write system required an average of ~0.0227ms +- 0.009ms per frame. The readonly system required an average of ~0.0156ms +- 0.0067ms. (See figure 61)

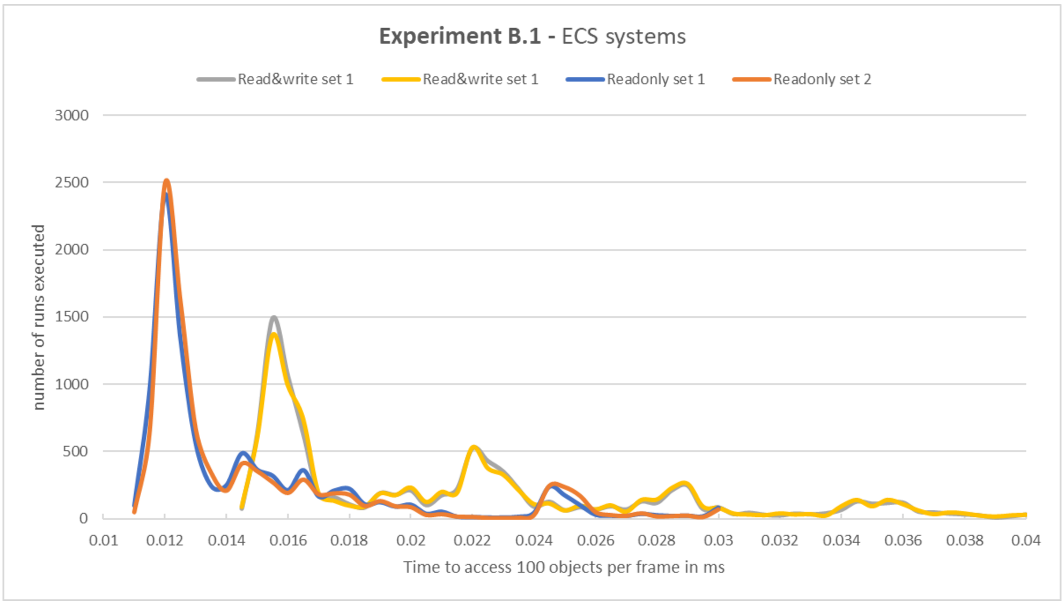


Figure 61 Results of experiment B.1 – ECS systems (Kyaw, 2022)

The structure of the data used for the ECS system was compared to the AoS implementation. Unfortunately, the AoS read-write system outperformed the ECS system by ~930% with ECS performing around 0.0205ms slower on average. The AoS readonly also outperformed the ECS system with ECS performing around 0.0146ms slower on average.

The given results concluded that ECS code does not execute faster than classic Unity code in regards to memory access speed. In fact, ECS code performed drastically worse than classic Unity code.

5.1.5 Evaluation: Experiment C

This experiment analyzed the capability of the DOTS renderer in comparison to the renderer of classic Unity. The render mesh used for the cubes consists of 96 vertices and 132 indices and was more complex than the average cube consisting of rounded edges (see figure 62). The results of the render performance are presented as a graph in fps (frames per second) per instanced units. The x axis describes the number of units instanced and the y axis describes the average fps during that moment. Each graph has a marker that indicates the moment that the frames dropped below the targeted 60 frames per second. (See figure 63)

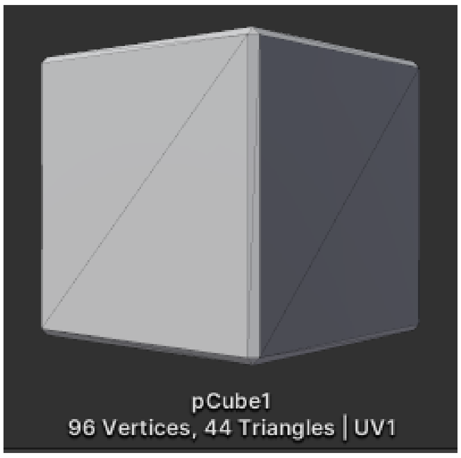


Figure 62 Experiment C – cube's render mesh data (Kyaw, 2022)

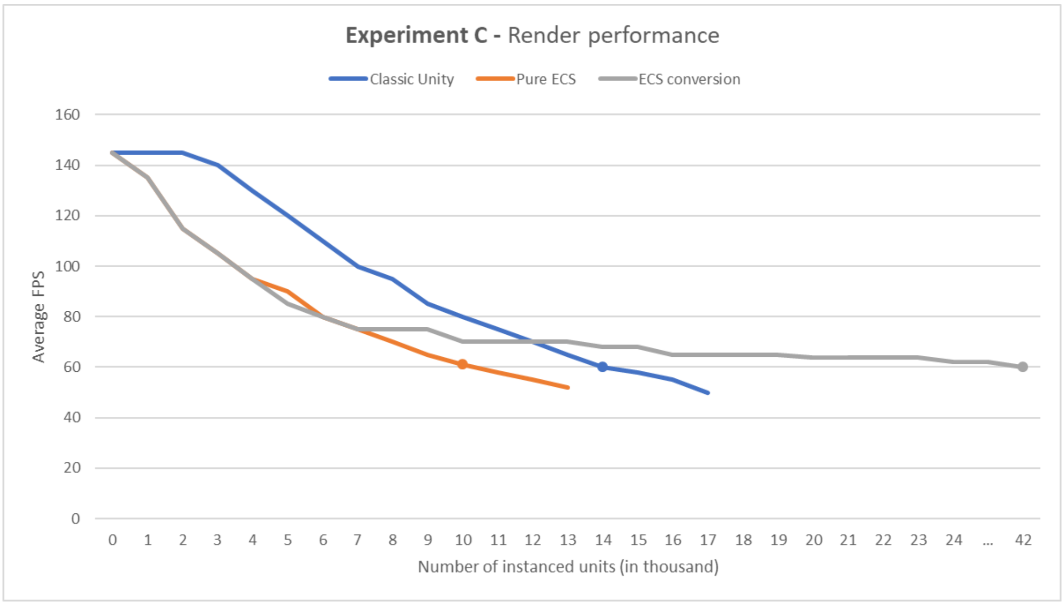


Figure 63 Results of experiment C - render performance (Kyaw, 2022)

All measured fps values of the render tests represent an approximation with some deviation. The results showed that pure ECS performed worse than the classic Unity implementation. ECS struggled to keep 60 fps rendering 10k cubes while classic Unity was able to hold 60 fps with 14k cubes rendered. These results show that classic Unity performed the same as ECS with 40% more cubes rendered. In contrast the Entity conversion performed the best by a huge margin. During first seven thousand cubes, pure ECS and entity conversion performed around the same. After that the entity conversion was able to hold its fps more stable than ECS or classic Unity until it reached around 42 thousand cubes rendered.

From the given observations the experiment evaluated that a handmade entity using pure ECS which was modeled after the conversion entity, did perform worse than the classic Unity gameobject. The reason for this outcome was potentially that the entity archetype used for the entity cubes or the ECS processing code created entities differently than conversion entities.
Conclusively, the hybrid renderer of Unity DOTS does perform better than the classic Unity renderer when using the ECS entity conversion system.

In addition, experiment B.2 was conducted in this experiment and aimed to perform load tests on the creation of game objects and entities including the entity conversion. The results of this experiment are presented in a graph whose x axis describes the time elapsed to instance 1000 units in milliseconds. The y axis shows the frequency of the occurrence of those times measured. (See figure 64)

The planned on-creating-conversion setting was implemented as a proof of concept but not further tested because of its extremely inefficient use of the conversion system and immense stress on the system.

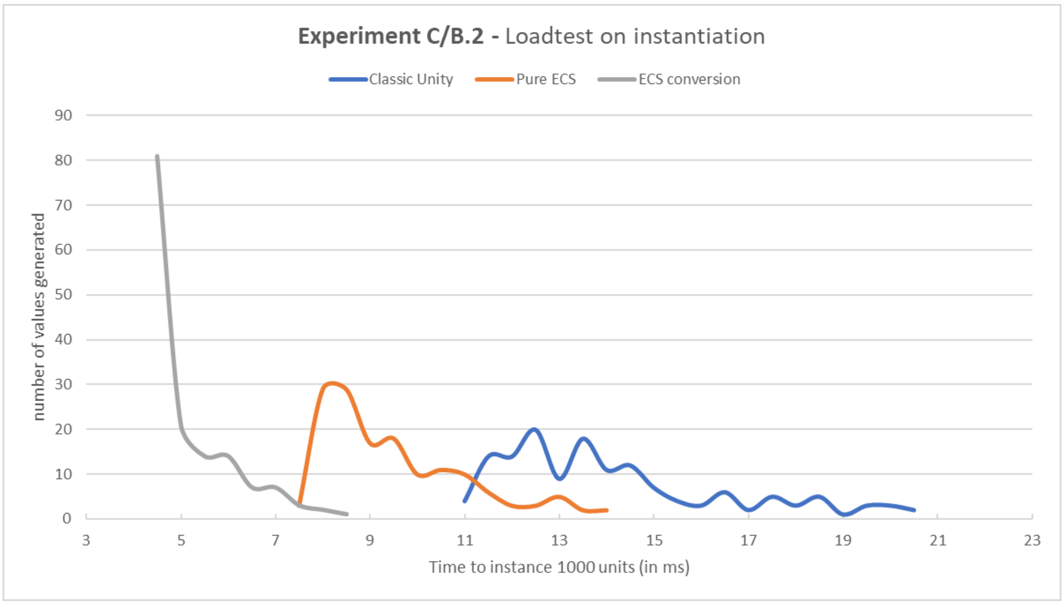


Figure 64 Results of experiment B.2 - load testing on the instantiation of gameobjects and entities (Kyaw, 2022)

The graph shows that the results of the load testing were similar to the render tests' results with strongly varying results between each method. The classic Unity method required an average time of ~14.2ms to instance 1000 units while the pure ECS method performed around ~30% faster with an average time of ~9.4ms. The entity conversion method outperformed both of the other methods by a margin with an average time of ~4.9ms.

5.1.6 Evaluation: Experiment D

This experiment analyzed the performance of the multithreading solution job system in comparison to classic Unity including basic C# threading by implementing merge sort algorithms. The results of this experiment are presented in multiple graphs whose x axis describe the time elapsed for the merge sort algorithm to sort an array in milliseconds. The y axis shows the frequency of the occurrence of those times measured.

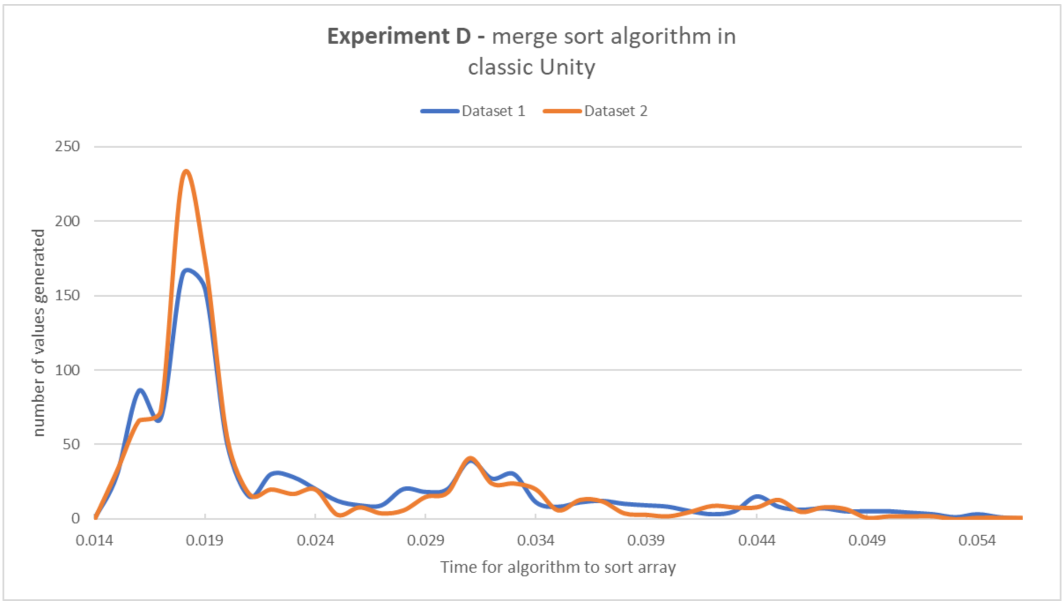


Figure 65 Results of experiment D - merge sort algorithm in classic Unity (Kyaw, 2022)

The classic Unity implementation required an average of ~0.025 ms to sort the array and acted as the control to evaluate the performance of the multithreading implementations. (See figure 65)

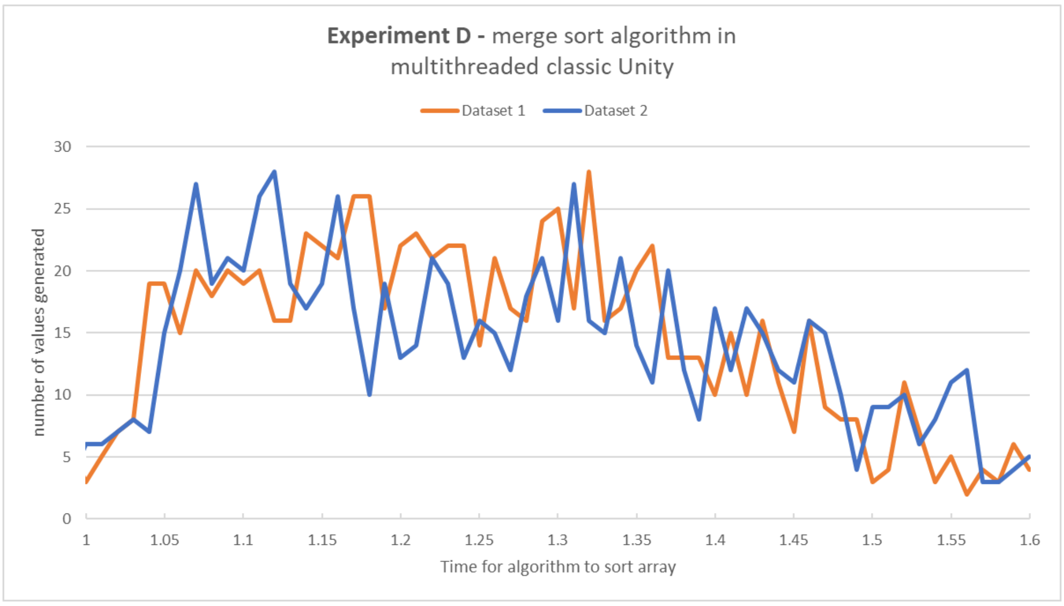


Figure 66 Results of experiment D - merge sort algorithm using C# threading (Kyaw, 2022)

In comparison the C# threading implementation resulted in ever spread-out time values while the classic Unity implementation had a more focused pool of results (see figure 66). The C# threaded algorithm required an average time of ~1.6ms to sort the array. Both datasets represented in the graph had the same average time but showed very little overlapping sections on the results which indicates that the execution of the algorithm is not consistent and strongly dependent on context switching and thread timing.

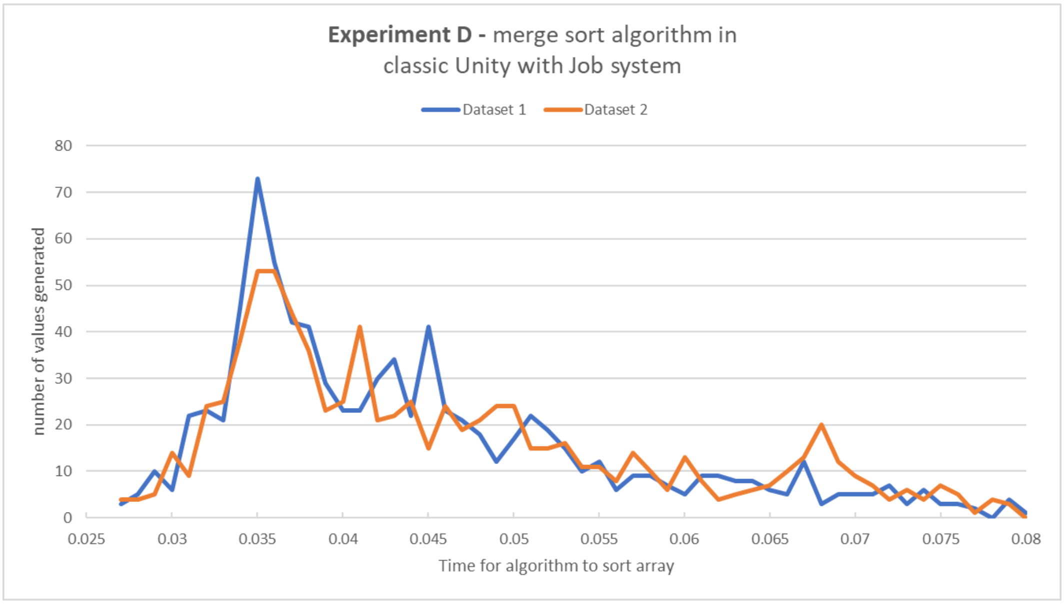


Figure 67 Results of experiment D - merge sort algorithm using job system (Kyaw, 2022)

The other multithreading implementation used the job system and showed way better performance than the basic C# threading with an average time of ~0.053ms to sort the array (see figure 67). Unfortunately, the job system implementation performed around ~50% slower on average than the classic Unity implementation as seen in the figure 68.

This experiment evaluated that the job system performs better than the C# threading with more consistent results because of the engine integration. The job system implementation performed worse than the non-multithreaded classic Unity version which was most likely to suboptimal implementation because of the complex nature of system.

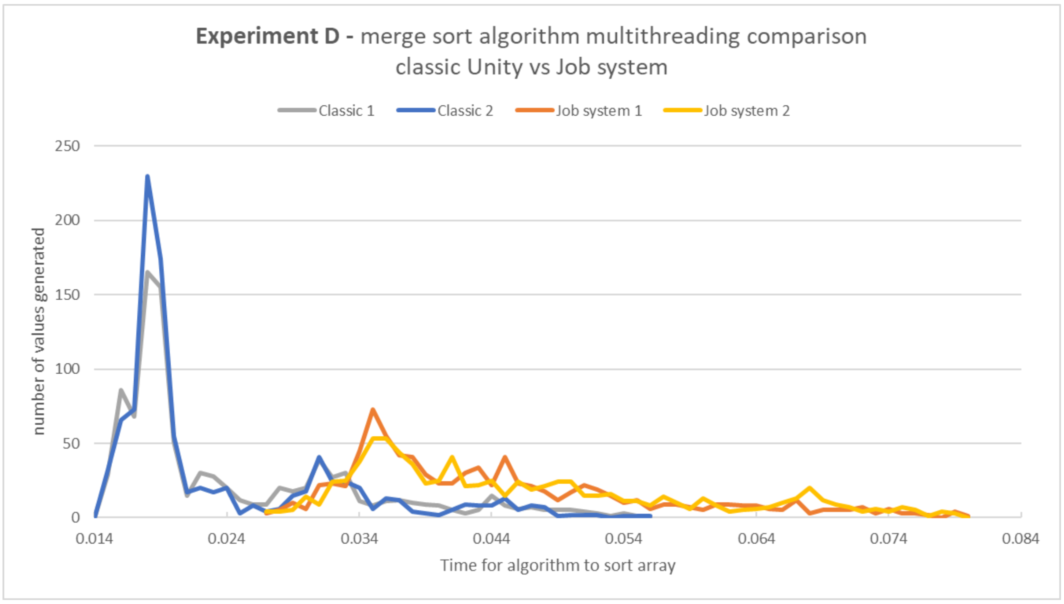


Figure 68 Results of experiment D – comparison classic Unity and Job system (Kyaw, 2022)

5.1.7 Evaluation: Experiment E

This experiment implemented a stress test that tests all of the various aspects analyzed in the previous experiments.

The cubes rendered were the same as from experiment C. The results of the render performance are presented as a graph in fps per instanced units. The x axis presents the number of units instanced and the y axis shows the average fps during that moment. A marker on the graphs indicates the moment the average fps dropped below 60 frames. (See figure 69)
The results of the performance of the movement system are also presented in graphs. The x axis describes the number of instanced units in thousands and the y axis describes the average time the system needs to execute per frame. (See figure 70,71)
All measured values of this experiment represent approximations with some minor deviation.

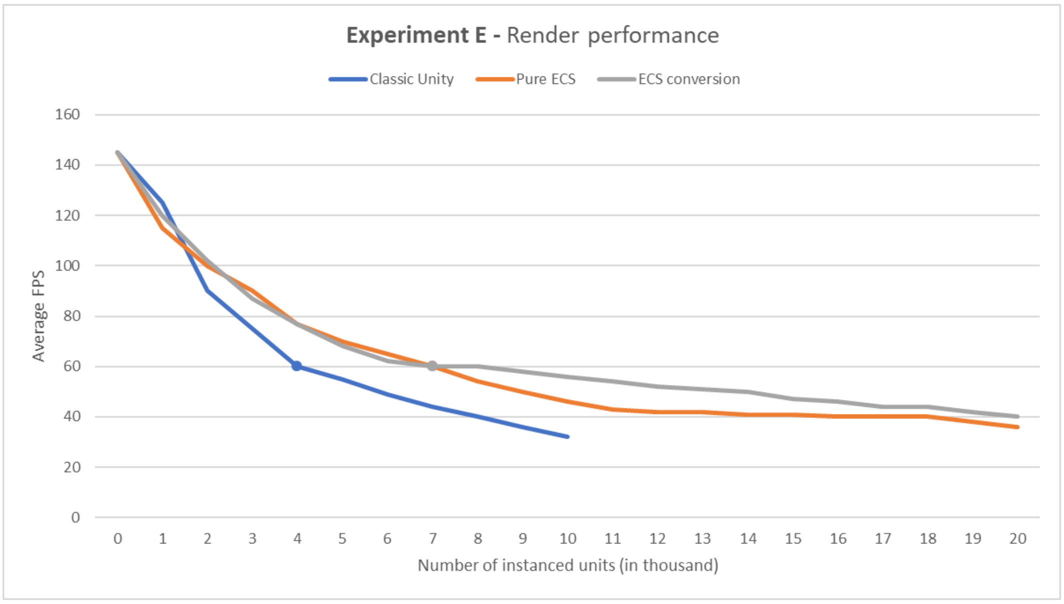


Figure 69 Results of experiment E - render performance (Kyaw, 2022)

The results of the render performance differed greatly from the results of experiment c. The pure ECS and entity conversion showed very similar results with both dropping below 60 fps at around 7000 units rendered while classic Unity already fell off at 4000 units. The results of this experiment's render performance in comparison to experiment c indicated that Unity DOTS hybrid renderer performs better in game like scenarios where a lot of render updates occurs.

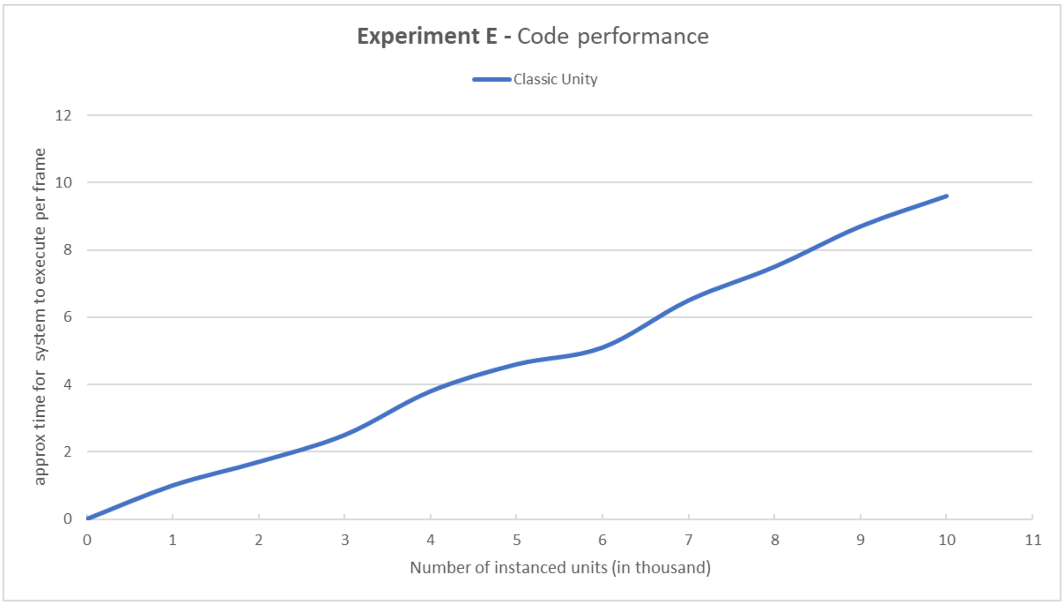


Figure 70 Results of experiment E - code performance of classic Unity movement system (Kyaw, 2022)

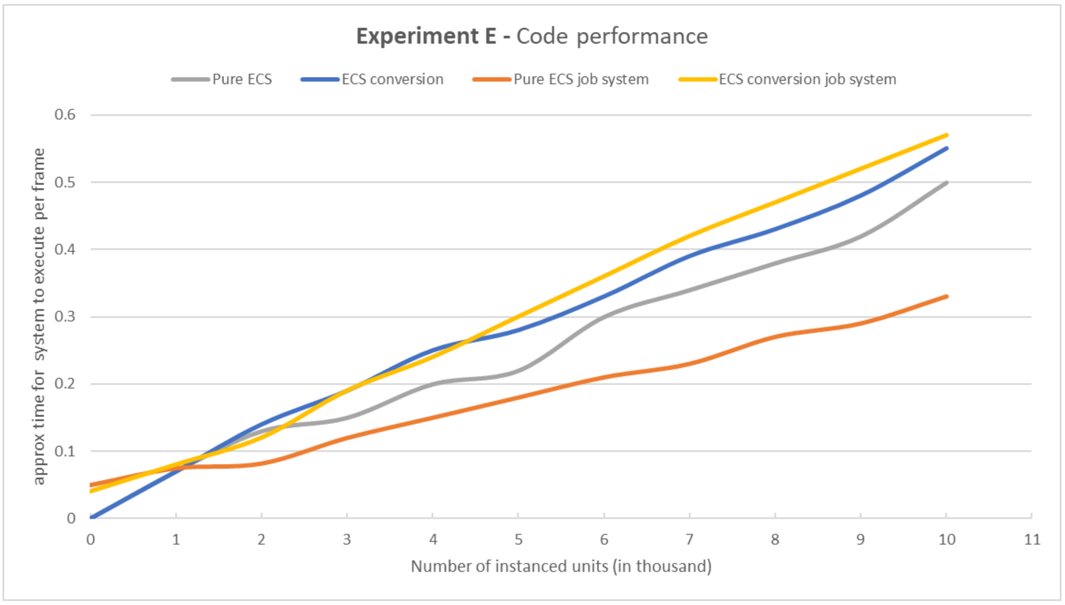


Figure 71 Results of experiment E - code performance of ECS movement systems (Kyaw, 2022)

The results of the classic Unity movement system showed considerably worse performance as the ECS system as expected. The classic Unity implementation required a movement system per instanced unit in contrast to the one system that updates all entities in ECS. The graphs in figure 71 show that most ECS systems performed at around the same performance with minimal difference except for the pure ECS with job system implementation. The pure ECS job performed around ~30 faster on average than the other ECS systems.

5.2 Conclusion

This chapter gave a conclusive statement on each feature of the Unity DOTS under consideration of the hypotheses made in this thesis through the analysis of the results given in 5.1 Evaluation.

5.2.1 Conclusion on ECS

The ECS as described during the context chapter offers a lot of more performant feature over the classic Unity OOP. One of the hypotheses regarding the ECS was the statement by Unity Technologies that it "[…] iterate[s] faster with C# code […]." *(Unity Technologies, no date b)*
From the results given in experiment B.1 it would possible to conclude that ECS does not perform better that classic Unity code. When comparing the results of experiment B.1 to experiment E it became apparent that ECS performs considerable better when regarding updating larger quantities of entities. Classic Unity required running an Update on every Monobehaviour script on every Gameobject. The ECS stores components into contiguous arrays and a single function in a system can define the behavior for thousands of similar entities. The fundamental way ECS operates on entities by letting the system only read the needed data for its calculation allows it to perform better on more complex and strenuous tasks.

Regarding performance of the ECS another hypothesis stated that a Unity DOTS project requires less memory during runtime.
From the results of experiment A.1 and A.2 its concluded that a project in Unity DOTS allocates less memory during runtime in comparison classic Unity projects.

Another aspect to consider regarding performance is the renderer performance with a hypothesis stating that the hybrid renderer renders polygons more efficiently than the classic Unity renderer.
Considering the results of experiment c and e it is concluded that the hybrid renderer is capable of rendering polygons more efficiently. The hybrid renderer can especially hold its fps more stable over a larger number of entities rendered.

When considering usability, a hypothesis stated that DOTS code is “[…] easier to read and reuse across other projects.” *(Unity Technologies, no date b).* In addition, another hypothesis stated that ECS makes the use of data-oriented design easier and more accessible.
The data-oriented principle of separating data and logic and composition over inheritance enforced through the ECS structure allowed for components and systems to be reused to other project without any complication. Components and system do not have any dependencies like inheritance, making the components and system independent and therefore way more flexible to change and reuse.
The application of the data-oriented principles is made very simple through the ECS structure inherently implementing those principles.

Conclusively, it is possible to say that the ECS architectural pattern and the Unity ECS offer a huge advantage over classic Unity regarding performance and usability. With Unity DOTS being an unfinished project, the ECS will likely improve overtime as seen with the newest major update of the 0.50 version which was released during the writing of this thesis.

5.2.2 Conclusion on OOP vs DOD/ECS

Another topic regarding ECS is the comparison between the use of OOP and DOD and its influence on performance through ECS.
In most cases it is regarded that code is an aspect of performance and not data. Through the analysis of data locality in experiment B.1 it became apparent that data has a considerable impact on performance. The memory optimized data struct SoA offered a performance improvement of ~200% over AoS during experiment B.1.

The data-oriented principle of data locality is utilized in most feature of Unity DOTS. The ECS structures its data by archetypes in memory to create contiguous data which allows systems to access them more efficiently. Using contiguous data is also heavily favored by the hardware especially the cache. The systems don’t look at an entity as an entirety instead only considers data allowing entities to be updated by archetype. These archetypes are stored in memory chunks only containing entities of the same archetype allowing for easier parallelization.

In general, applying principle of DOD and ECS will heavily improve the performance of any code. Principles such as using simple data structures like struct, keeping hierarchies flat and applying data locality can be used in nearly any programming language. Unfortunately, ECS is vastly different to OOP making it harder for Unity developers to switch to.
„If you think of Entities/Component's in OOP terms, you will never understand the ES, and you will screw up any attempt to program with it. […] It's fundamentally different and incompatible." *(Adam, 2007)*

5.2.3 Conclusion on the C# Job system

Multithreading is a technology that gained more and more importance over the recent years. So, Unity Technologies stated that the "[…] Data-Oriented Technology Stack (DOTS) will." *(Unity Technologies, no date a)* More specifically the multithreading solution C# job system.

Multithreading in general offers a lot of advantages which unfortunately comes with quite a few complications. Multithreading allows for a better use of the system's resources by using threads to execute operations sequentially. Using more threads in turn means increasing complexity and more complex synchronization of shared data. Without proper synchronization the threads are strongly dependent on timing and context switching. Multithreading is generally difficult to debug.

Now comes the question whether the job system does solve these issues with minimal programming headache. During this thesis the multithreading solution C# job system was implemented in two different ways.

The first implementation was executed in experiment D with the job system implemented in classic Unity as threading for an algorithm. Implementing the job system without the ECS resulted in various problems. Each job required a previous job’s results to schedule resulting in the none multithreaded classic Unity implementation performing around ~50% faster than the job implementation. Because of the nature of the algorithm a lot of jobs needed to be scheduled manually with each requiring dependencies to previous jobs which made the implementation very disorganized.

The second implementation worked in conjunction with the ECS to multithread systems which is the primary use of the job system. In order to multithread a system only one key word needed to be changed. The Run method needed to be replaced with the ScheduleParallel method. In the case of experiment E using the job system offered an improvement of around ~30% in comparison to the non-multithreaded ECS system.

Fortunately, Unity Technologies stated that Unity DOTS as an entirety would make it possible to take full advantage of multicore processors without the heavy programming headache. Having to change a single keyword in order for an ECS system to utilize multithreading without worries about multithreading problems can definitely be described as “without programming headaches”.

Conclusively as seen by the implementation of experiment D the job system should not be used without the ECS to multithread algorithms. This method can potentially be very complicated and complex to implement. The job system does not seem to be intended for complex asynchronous computation. Utilizing the job system with ECS offers an immense improvement to performance with minimal effort.

5.2.4 Conclusion on the usability of the Unity DOTS packages

When considering the aspect of usability, the ease of learning to develop using the packages is also considered which include finding data and examples on the matter.
Currently Unity DOTS has a huge learning curve because of its lack of up do date guide on basic implementation structure and more advanced feature. The Unity documentation is currently also not complete and some pages do not offer a detail enough explanation on implementation.

With Unity DOTS being a relatively new system finding information on certain problems and features is pretty difficult because of the different syntax between versions and deprecated features. An example for this problem was during the implementation of experiment B.1 with creating array like structure in components. Most sources on this feature present complex implementation for other uses that were not applicable to experiment B.1.

Besides the huge learning curve Unity DOTS offers a feature called entity conversion which simplifies the use of entities and in most cases results in better performance than hand made ECS entities. In the case of experiment C, the entity conversion approach offered a better creating time and more efficient rendering result. The entity conversion in general offers the use of ECS without the potential risk of inefficient coding.

5.2.5 Final thoughts

Under consideration of the conclusions, this thesis can conclude that learning ECS and data-oriented principles is beneficial to the development of future project and general programming understanding. Although the use of concepts and principles from data-oriented design is recommended, it is not fully recommended to develop games with Unity DOTS yet. As previously described, Unity DOTS has a major learning curve with few up to date information on developing with DOTS. With each update previously established syntax or methods could and have become deprecated. In addition, Unity DOTS is currently released as an experimental package in the Unity Engine which indicate that there are potential flaws in the system.

In the current stated of Unity DOTS, it is advised to develop games with caution. Another analysis would be beneficial when Unity DOTS is released as a release version. All experiment results could have resulted differently when the programs were written better but from the given time frame and resources found the experiment s were created to best abilities of the programmer.

Appendix

ResearchProjectDOTS folder

This folder contains the source code for the experiments conducted during this thesis

ResearchProjectDOTS/Packages/manifest.json

This json file contains the version of all packages used within the Unity project for the experiments.

Experiment-Results.xlsx

This excel file contains the data collected and graphs for the evaluation section.

References

Adam (2007) Entity Systems are the future of MMOG development – Part 2. [online] <http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/> [accessed 20.01.2022]

Avgeriou, Paris/ Zdun, Uwe (2005) Architectural patterns revisited - a pattern language. Universitaetsverlag Konstanz.

Baumann, Eric (no date) A recursive merge sort algorithm used to sort an array of 7 integer values. These are the steps a human would take to emulate merge sort (top-down). [online] <https://en.wikipedia.org/wiki/Merge_sort#/media/File:Merge_sort_algorithm_diagram.svg> [accessed: 20.01.2021]

Bilas, Scott (2002) A Data-Driven Game Object System – GDC talk 2002. [online] <https://www.gamedevs.org/uploads/data-driven-game-object-system.pdf> [accessed: 31.05.2022]

Cheney, Dave (2014) The acme of foolishness - Five things that make Go fast. [online] <https://dave.cheney.net/2014/06/07/five-things-that-make-go-fast> [accessed: 17.11.2021], graph

Choudhary, Pankaj Kumar (2015) Understanding Unsafe Code in C# [online] <https://www.c-sharpcorner.com/UploadFile/f0b2ed/understanding-unsafe-code-in-C-Sharp/> [accessed: 20.12.2021]

Doucet, Lars/ Pecorella, Anthony (2021) Game engines on Steam: The definitive breakdown. [online] <https://www.gamedeveloper.com/business/game-engines-on-steam-the-definitive-breakdown> [accessed: 01.07.2022]

ECMA, International (no date) ECMA-334 - C# language specification. [online] <https://www.ecma-international.org/publications-and-standards/standards/ecma-334/> [accessed: 12.11.2021]

Edpresso (no date) Contiguous memory. [online] <https://www.educative.io/edpresso/contiguous-memory> [accessed 20.11.2021]

Entity Systems Wiki (2014) What's an Entity System? [online] <http://entity-systems.wikidot.com> [accessed: 30.01.2022]

Garfinkel, Simson L. (2005) Design Principles and Patterns for Computer Systems That Are Simultaneously Secure and Usable. PhD thesis. Massachusetts Institute of Technology. [online] <https://simson.net/thesis/> [accessed: 15.11.2021]

Garlan, David/ Shaw, Mary (1994) An Introduction to Software Architecture. School of Computer Science Carnegie Mellon University Pittsburgh

Gamma, Erich/ Helm, Richard/ Johnson, Ralph/ Vlissides, John (1995) Design Patterns: Elements of Reusable Object-Oriented Software. Massachusetts: Addison-Wesley. ISBN 0-201-63361-2.

GeeksforGeeks (2021) Deadlock in Java Multithreading [online] <https://www.geeksforgeeks.org/deadlock-in-java-multithreading/> [accessed: 20.02.2022]

GeeksforGeeks (2019) Multithreading in Phyton | Set 2 (Synchronization). [online] <https://www.geeksforgeeks.org/multithreading-in-python-set-2-synchronization/> [accessed: 20.02.2022]

Hayes, Adam (2022) Descriptive Statistics Definition. [online] <https://www.investopedia.com/terms/d/descriptive_statistics.asp> [accessed: 20.07.2022]

Hollmann, Trevor (2019) Development of an Entity Component System Architecture for Realtime Simulation. Bachelor thesis. Koblenz. [online] <https://kola.opus.hbz-nrw.de/opus45-kola/frontdoor/deliver/index/docId/1932/file/thesis-2019-09-16-final.pdf> [accessed: 24.11.2021]

Holopainen, Timo (2016) Object-oriented programming with Unity - Inheritance versus composition. Bachelor thesis. [online] <https://www.theseus.fi/bitstream/handle/10024/108967/Holopainen_Timo.pdf?sequence=1> [accessed: 08.11.2021]

Intel Corporation (2018) How to Manipulate Data Structure to Optimize Memory Use on 32-Bit Intel® Architecture. [online] <https://www.intel.com/content/www/us/en/developer/articles/technical/how-to-manipulate-data-structure-to-optimize-memory-use-on-32-bit-intel-architecture.html> [accessed: 20.11.2021]

Intel Corporations (2019) Memory Layout Transformations. [online] <https://www.intel.com/content/www/us/en/developer/articles/technical/memory-layout-transformations.html> [accessed: 20.11.2021]

ISO/IEC (2018) ISO/IEC23270 - Information technology - Programming languages - C#. Switzerland

ISO (2018) Ergonomics of human-system interaction — Part 11: Usability: Definitions and concepts. [online] <https://www.iso.org/obp/ui/#iso:std:iso:9241:-11:ed-2:v1:en> [accessed: 31.05.2022]

Johansson, Tim (2018) Job System & Entity Component System. Game Developers Conference – GDC in San Fransisco. San Fransico 23th March.

Available at: https://www.youtube.com/watch?v=kwnb9Clh2Is&t=1s [accessed: 20.01.2022]

Klutzershy (2013) Understanding Component-Entity-Systems. [online] <https://www.gamedev.net/tutorials/_/technical/game-programming/understanding-component-entity-systems-r3013/> [accessed: 31.01.2022]

Knoernschild, Kirk (2001) Java Design: Objects, UML, and Process. Amsterdam: Addison-Wesley Longman ISBN-: 978-0201750447

Leavens, Gary T. (1998) Homework Web Pages for Com S 541 – Paradigms – First part – Programming Paradigms. [online] <https://www.cs.ucf.edu/~leavens/ComS541Fall98/hwpages/paradigms/paradigms.htm> [accessed: 11.11.2021]

Lewallen, Raymond (2005) 4 major principles of Object-Oriented Programming. [online] <http://codebetter.com/raymondlewallen/2005/07/19/4-major-principles-of-object-oriented-programming/> [accessed: 09.11.2021]

Llopis, Noel (2009) Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP). [online] <https://gamesfromwithin.com/data-oriented-design> [accessed 18.11.2021]

Losonczi, Tamás (2020) Introduction to Data-Oriented Programming - A shift from objects to data. [online] <https://medium.com/mirum-budapest/introduction-to-data-oriented-programming-85b51b99572d> [accessed: 22.11.2021]

Roathe, Lane (2010) OOP != classes, but may == DOD. [online] <https://roathe.wordpress.com/2010/03/22/oop-classes-but-may-dod/> [accessed 18.11.2021]

Sam, Samuel (2020) Deadlock in Java Multithreading [online] <https://www.tutorialspoint.com/Deadlock-in-Java-Multithreading> [accessed: 31.01.2022]

Subburaj, Parthasarathy (2020) Exploiting Multiprocessing and Multithreading in Python as a Data Scientist [online] <https://medium.com/analytics-vidhya/exploiting-multithreading-and-multiprocessing-in-python-as-a-data-scientist-e2c98b61997a> [accessed: 20.01.2022]

Toftedahl, Marcus (2019) Which are the most commonly used Game Engines? [online] <https://www.gamedeveloper.com/production/which-are-the-most-commonly-used-game-engines-> [accessed: 15.10.2021]

Udacity (2015) Scatter To Gather Transformation - Intro to Parallel Programming. [online] <https://www.youtube.com/watch?v=AcULfKtWfoA> [accessed: 20.11.2021]

Unity Documentation (no date a) ECS Concepts [online] <https://docs.unity3d.com/Packages/com.unity.entities@0.50/manual/ecs_core.html> [accessed: 26.11.2021]

Unity Documentation (no date b) Entities [online] <https://docs.unity3d.com/Packages/com.unity.entities@0.50/manual/ecs_entities.html> [accessed: 30.11.2021]

Unity Documentation (no date c) Conversion Workflow [online] <https://docs.unity3d.com/Packages/com.unity.entities@0.50/manual/conversion.html> [accessed: 20.12.2021]

Unity Documentation (no date d) Burst User Guide [online] <https://docs.unity3d.com/Packages/com.unity.burst@0.2/manual/index.html> [accessed: 24.03.2022]

Unity Documentation (no date e) Unity.Mathematics [online] <https://docs.unity3d.com/Packages/com.unity.mathematics@1.0/manual/index.html> [accessed: 24.03.2022]

Unity Documentation (2018) C# Job System [online] <https://docs.unity3d.com/Manual/JobSystem.html> [accessed: 20.01.2022]

Unity Technologies (no date a) DOTS Packages. [online] <https://unity.com/dots/packages#entities-preview> [accessed: 14.10.2021]

Unity Technologies (no date b) DOTS - Unity's new multithreaded Data-Oriented Technology Stack. [online] <https://unity.com/dots> [accessed: 14.10.2021]

Unity Technologies (2018) Megacity Demo. [online] <https://unity.com/megacity> [accessed: 08.09.2021]

Unity Technologies (2019) Unite Conference Copenhagen – Video. [online] <https://youtube.com/playlist?list=PLX2vGYjWbI0S1wHRTyDiPtKLEPTWFi4cd> [accessed: 26.10.2021]

Unreal Engine (no date) Unreal Engine 5. [online] <https://www.unrealengine.com/en-US/unreal-engine-5> [accessed: 22.11.2021]

Martin, Robert C. (2000) Design Principles and Design Patterns. [online] <https://web.archive.org/web/20120115094537/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf> [accessed: 11.11.2021]

Martin, Robert C. (no date) The Principles of OOD [online] <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod> [accessed: 22.11.2021]

McMurray, James (2020) An introduction to Data Oriented Design with Rust. [online] <https://jamesmcm.github.io/blog/2020/07/25/intro-dod/> [accessed: 19.11.2021]

Fuad, Matt (2022) Experimental Entities 0.50 is available [online] <https://forum.unity.com/threads/experimental-entities-0-50-is-available.1253394/> [accessed: 13.04.2022]

Microsoft (2021 a) override (C# reference). [online] <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/override> [accessed: 14.11.2021]

Microsoft (2021 b) Methods (C# Programming Guide). [online] <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/methods> [accessed: 14.11.2021]

Nikolov, Stoyan (2018) CppCon 2018- Data-oriented design in practice. [online] <https://github.com/CppCon/CppCon2018/blob/6eafc8726e338becfef05bfd0a3508759cd62278/Presentations/oop_is_dead_long_live_dataoriented_design/oop_is_dead_long_live_dataoriented_design__stoyan_nikolov__cppcon_2018.pdf> <https://www.youtube.com/watch?v=yy8jQgmhbAU> [accessed 18.11.2021]

Nørmark, Kurt (2014) Functional Programming in Scheme with Web Programming Examples. [online] <http://people.cs.aau.dk/~normark/prog3-03/html/notes/theme-index.html> [accessed: 11.11.2021] Denmark: Aalborg University

Nystrom, Robert (no date) Data locality – Game Programming Patterns/ Optimization Patterns. [online] <http://gameprogrammingpatterns.com/data-locality.html> [accessed: 19.11.2021]

Van Roy, Peter (2009) Programming paradigms for Dummies: What every programmer should know. Belgium: Université Catholique de Louvain - UCLouvain

West, Mike (2007) Evolve Your Hierarchy. [online] <https://web.archive.org/web/20190116045950/http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/> [accessed: 24.11.2021]

Yu, Haiying (2022) Race conditions and deadlocks [online] <https://docs.microsoft.com/en-GB/troubleshoot/developer/visualstudio/visual-basic/language-compilers/race-conditions-deadlocks> [accessed: 26.05.2022]

Bibliography

Acton, Mike (2014) Data-Oriented Design and C++ - CppCon 2014 [online] <https://www.youtube.com/watch?v=rX0ItVEVjHc> [accessed: 19.11.2021]

Brooks Jr., Frederick P. (1986) No Silver Bullet – Essence and Accident in Software Engineering [online] <http://worrydream.com/refs/Brooks-NoSilverBullet.pdf> [accessed: 19.11.2021]

Fabian, Richard (2018) Data-oriented design: software engineering for limited resources and short schedules, ISBN-13 978-1916478701 <https://www.dataorienteddesign.com/dodmain/node1.html> <https://www.dataorienteddesign.com/dodbook/> [accessed: 19.11.2021]

GeeksForGeeks (2019) Difference between Multiprogramming, multitasking, multithreading and multiprocessing [online] <https://www.geeksforgeeks.org/difference-between-multitasking-multithreading-and-multiprocessing/> [accessed: 20.01.2022]

Gillis, Alexander S. (2021) Definition object-oriented programming (OOP) [online] <https://searchapparchitecture.techtarget.com/definition/object-oriented-programming-OOP> [accessed: 12.11.2021]

Hodgman (2018) OOP is dead, long live OOP [online] <https://www.gamedev.net/blogs/entry/2265481-oop-is-dead-long-live-oop/> [accessed: 20.11.2021]

Nirosh (2021) Introduction to Object Oriented Programming Concepts (OOP) and More [online] <https://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep> [accessed: 08.11.2021]

Nystrom, Robert (2014) Game Programming Patterns, genever benning, ISBN: 978-0-9905829-0-8

Pranckevičius, Aras (2018) dod-playground [online] <https://github.com/aras-p/dod-playground> [accessed 28.11.2021]

Pup (2015) Is the Entity Component System architecture object oriented by definition? [online] <https://softwareengineering.stackexchange.com/a/279545> [accessed 31.05.2022]

The Latency Elephant (2009) [online] <http://seven-degrees-of-freedom.blogspot.com/2009/10/latency-elephant.html> [accessed: 21.11.2021]

Object Mentor (2006) What Is Object-Oriented Design? [online] <https://web.archive.org/web/20070630045531/http://www.objectmentor.com/omSolutions/oops_what.html> [accessed: 14.11.2021]