Bachelor's thesis

Degree Programme in Information and Communications Technology

2021

Alarik Näykki

UNITY DOTS IN PRODUCTION

DOTS pathfinding implementation in VR & AR.

TURKU AMK
TURKU UNIVERSITY OF
APPLIED SCIENCES

Bachelor's | Abstract

Turku University of Applied Sciences

Degree Programme in Information and Communications Technology

2021 | Number of pages 68

Alarik Näykki

Unity DOTS in production

- DOTS pathfinding implementation in VR & AR.

Unity's Data-oriented technology stack (DOTS) is Unity's approach to Data-oriented design in Unity. DOTS promises great performance gains compared to the current object-oriented Unity game development. DOTS is still in preview, which gives reason to research its current capabilities. This thesis aimed to find out if DOTS was ready to be used in production. The secondary goal of the thesis was to test the combability of VR and AR with DOTS.

The primary method used in this thesis was the implementation of pathfinding using DOTS. This pathfinding was benchmarked on two computers to compare the difference hardware makes in DOTS development. The pathfinding was also implemented into VR and AR environments to find any combability problems with DOTS. The pathfinding method was also implemented using the object-oriented Unity tools to compare the performance difference between it and the DOTS approach.

The pathfinding implementation was successful and ran up to 13 times faster compared to the object-oriented approach. Moreover, no combability issues with VR or AR with DOTS were found during the development. Still, it was concluded that DOTS is not ready to be used in production as its usage was deemed too arduous in its current state.

Keywords:

Unity, game development, data-oriented design, DOTS, C#, multithreading, VR, AR

Opinnäytetyö (AMK) | Tiivistelmä

Turun ammattikorkeakoulu

Tieto- ja viestintätekniikka

2021| 68 sivua

Alarik Näykki

Unity DOTS tuotannossa

- DOTS polunetsinnän toteuttaminen VR:ssä ja AR:ssä.

Unityn datasuuntautunut teknologiapino (DOTS) on Unityn lähestymistapa datasuuntautuneeseen suunnitteluun Unityssä. Unity lupaa suuria suorituskykyhyötyjä DOTS-kehityksessä nykyiseen oliosuuntautuneeseen Unity-pelikehitykseen verrattuna. DOTS on vielä esikatseluvaiheessa, joka tarkoittaa ettei se ole vielä valmis. Tämä antaa syyn tutkia sen nykyisiä ominaisuuksia. Tämän työn tavoitteena oli selvittää, onko DOTS valmis käytettäväksi tuotannossa. Tutkimuksen toissijainen tavoite oli testata VR:n ja AR:n yhteensopivuutta DOTSin kanssa.

Tärkein tutkimuksessa käytetty menetelmä oli polunetsinnän toteuttaminen DOTSin avulla. Tätä polunetsintää vertailtiin kahdella tietokoneella, jotta voitiin verrata laitteiston aiheuttamaa suorituseroa DOTS-kehityksessä. Polunetsintä toteutettiin myös VR- ja AR-ympäristöissä. Näin pyrittiin löytämään mahdolliset yhteensopivuusongelmat, joita DOTSilla saattaa olla VR:n tai AR:n kanssa. Polunetsintä toteutettiin myös oliosuuntautuneilla Unity-työkaluilla, jotta voitiin verrata sen ja DOTS-lähestymistavan välistä suorituskykyeroa.

Polunetsintätoteutus onnistui hyvin, ja se oli jopa 13 kertaa suorituskykyisempi verrattuna oliosuuntautuneeseen lähestymistapaan. Kehityksen aikana ei havaittu yhteensopivuusongelmia DOTS-kehityksessä VR:n tai AR:n kanssa. Silti pääteltiin, että DOTS ei ole valmis käytettäväksi tuotannossa, koska sen käyttöä pidettiin nykytilassaan liian hankalana.

Asiasanat:

Unity, pelin kehitys, data orientoitunut malli, DOTS, C#, monisäkeisyys, virtuaalinen todellisuus, lisätty todellisuus

CONTENTS

FIGURES

TABLES

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
AR	Augmented Reality
CPU	Central Processing Unit
DOD	Data-oriented Design
DOTS	Data-oriented Technology Stack
ECS	Entity Component System
FPS	Frames Per Second
GPU	Graphics Processing Unit
OO	Object-oriented
OOP	Object-oriented Programming
OS	Operating System
PC	Personal Computer
RAM	Random Access Memory
VR	Virtual Reality

1 INTRODUCTION

The data-oriented technology stack (DOTS) is Unity's approach to data-oriented design (DOD) for Unity development. Unity markets this new technology as a solution with significant performance benefits compared to the current development model. Their tagline is "Performance by default", which means that the developer can have excellent performance by default by following their data-oriented design patterns. However, DOTS has not been declared production-ready by Unity (Unity 2021b). Production-ready, in this case, means that the software is capable of fully meeting the requirements of developing videogames. Therefore, it is unknown when or if game developers will start using DOTS or what their opinion on it is. The author of this thesis considers the current situation of interest since it seems possible to investigate the readiness of the product before it is deemed officially ready by Unity and other developers. Therefore, the purpose of this thesis was to find out if DOTS was production-ready. Production readiness is evaluated by developing a pathfinding method in DOTS and comparing it to a similar method created using conventional Unity tools. This way, the production readiness can be discerned from performance measurements and experience gained during the development.

In addition, the AR and VR combability with DOTS are also a subject of the research, as the research done expressly on these combinations was limited. Both VR & AR are also greatly dependant on the hardware they run on, which gave the author more reason to research optimizations directly.

For the sake of conciseness, this thesis does not dive deep into conventional object-oriented Unity development topics but does compare some object-oriented topics to their data-oriented counterparts. Therefore, to best understand this thesis, the reader would need to have some existing understanding of object-oriented programming and Unity development

1.1 Structure of the thesis

The structure of the thesis is as follows. Chapter 2 introduces Data-oriented design and its motives. Chapter 3 explains Unity DOTS in detail. Chapter 4 explores past work on the topic and explains the hypothesis of this work. Chapter 5 goes through the methods used for the research. Chapter 6 presents the results of the methods. Chapter 7 discusses the findings, and finally, Chapter 8 concludes the research.

2 DATA-ORIENTED DESIGN

The term Data-oriented design was coined for game development by Llopis (2009) but has been around previously in various forms (Cardelli 1988; Joshi 2007). Llopis identified problems with how hard it was to obtain optimal performance using OOP, which led them to explore parallelization and cache utilization with DOD. Additionally, their article compares the advantages and drawbacks of DOD to OOP (Object-oriented programming) in game development.

Programming paradigms classify programming languages based on their features. Data-oriented design, or DOD for short, is an optimization technique (Fabian 2018). DOD is not defined as a paradigm, but it is helpful to discuss it in the context of paradigms. Common paradigms include object-oriented, procedural, and functional programming. A coding language can use multiple paradigms like C++ or focus on one like C# with object-oriented programming (Microsoft 2021b). Unity and Unreal are the most popular game engines (Doucet et al., 2021), and object-oriented programming has been the default paradigm for them (Unreal 2021; Unity 2021d). This is one of the reasons why DOD is often compared to object-oriented programming in a game development context.

Programs are about transforming data from one form to another (Acton 2014). A game, in essence, is a program that manipulates data to serve the interests of the player. So, it makes sense that in game development, we need to think about the data and its characteristics before designing the code that executes the transformation. First, the data needs to be understood to understand its problems. If the problem is understood, the cost of the problem is also understood. The cost of the problem can be reasoned with if the hardware can be understood. With this foundation, DOD aims to solve problems case-by-case rather than being as general as possible. Instead of solving the most generic problems, it solves the most common problems first. In the coming parts, the ways that DOD adheres to these principles are discussed.

2.1 Motives behind Data-oriented design

This section discusses the motives behind data-oriented design. Performance is the most significant motive behind DOD, which it tackles with cache utilization and parallelization. The secondary motive is the ability to create modular and easier to read code.

2.1.1 Cache Utilization

CPUs have outpaced RAM when it comes to speed over the years (Hennessy et al., 2017). If the memory cannot feed data to the CPU fast enough, it can create a performance bottleneck. A tool to combat this is the cache memory located on the CPU or close to it. The cache is faster than the main memory (i.e., RAM) but smaller in size. The cache's job is to handle data that needs to be processed frequently. Modern CPUs utilize three levels (i.e., L1, L2, L3) of cache memory.

Table 1 – Size and latency for typical desktop PC's CPU and RAM.

	L1	**L2**	**L3**	**Main Memory**
Size	64 KB	256-512 KB	8-64 MB	4-64 GB
Latency	1 ns	3-10 ns	10-20 ns	50-100 ns

Table 1 lists the size and latency of memory in typical desktop PCs (Hennessy et al., 2017). L1 cache is up to 100 times faster than the main memory. Likewise, the L2 is up to 33 times faster and the L3 up to 10 times. Thus, we can infer that using the cache memory to its fullest can bring significant performance gains.

DOD Organizes the data in sequential blocks. With these blocks, the data can be processed in chunks. Cache's job is to process the same data fast and frequently, and by organizing the data, it can do this. Examples of caches utilization are discussed in the upcoming parts.

2.1.2 Parallelization & multithreading

Parallelization of code means that multiple tasks can be processed concurrently. With multicore processors, this means processing the tasks on multiple cores at the same time. This way, the code becomes multithreaded. As a result, the developer can access all the cores equally with multithreading instead of loading most of the work onto the main thread.

The number of CPU cores available to the average PC gamer increases over time (Steam 2021). Following are some relevant statistics enforcing this from the Steam hardware surveys from April to August of 2021:

- **6-core** CPU usage had risen by **2.98% to 34.09%** of total usage
- **8-core** CPU usage had risen by **0.44% to 14.04%** of total usage
- **4-core** CPU usage had dropped by **1.77% to 36.99%** of total usage
- **2-core** CPU usage had dropped by **1.63% to 11.96%** of total usage

We can convey from these statistics that CPUs with more than four cores are becoming more popular over time. The popularity is most likely due to midrange CPUs getting more cores on average. For example, the new & popular (PassMark Software 2021a) midrange AMD Ryzen 5 5600X series of CPUs have six cores. When the consumers have access to more cores, it will also be useful for the developer to know how they can utilize them in the future.

2.1.3 Modularity

DOD also has advantages when it comes to code readability and usability. Modularity is the major one. In DOD, the data is independent of its function. This means that functions do not depend on other parts of code. As a result, the code is easier to reuse as it is less reliant on its environment, making future development faster.

2.2 Differences between OOP and DOD

With DOD, data comes first. In DOD, data is handled as it is needed. In OOP, it is typical to try and create classes that are as generic and abstract as possible so that they can be used in as many situations as possible. Creating generics might sound like a good idea, but it can easily create bloated classes in practice. In DOD, only the data that is needed is created and used. The data is created to account for the most common occurrences instead of the most generic ones.

2.2.1 OOP Problem example

In order to illustrate the type of problems that OOP brings when used in games, a typical C# OOP class is presented next, see Figure 1. These classes use inheritance, allowing the class to inherit the features and variables of the class it inherits. First, the base class `Unit` has two variables, named `position` and `alive`. These two variables are something that any of the following units should have. After that, the `EnemyUnit` class is created with variables and methods that it can use to interact within a typical combat scenario. In this case, the specialized `EnemyUnits` can attack at range or fly. `RangerEnemy` and `FlyingEnemy` are created for that purpose.

```
class Unit
{
	Vector3 position;
	bool alive;
}
class EnemyUnit : Unit
{
	int maxHp;
	int currentHp;
	int attackPower;
	void Attack() { }
	void TakeDamage() { }
	void Move(Vector3 direction) { }
}
class RangerEnemy : EnemyUnit
{
	int attackRange;
}
class FlyingEnemy : EnemyUnit {}
```

Figure 1 – OOP class example 1.

This hierarchy of code has created several problems, which are covered next. What would happen if we wanted to have a destroyable tree unit? The tree does not need to attack or move. We conclude that we need to have a smaller jump in complexity between `Unit` and `EnemyUnit`. So we create the `CombatUnit` class with only the needed variables and methods for the tree.

Our next problem is that we want an enemy that can attack at range and fly. We have `RangedEnemy` and `FlyingEnemy`. We cannot create a class that inherits them both as they both already inherit `EnemyUnit`. Should we add these values to the `EnemyUnit`? Should we create a new class called `FlyingRangerEnemy` with both variables? For the sake of example, we create a new class, `SpecialEnemyUnit,` to hold variables that a specialized enemy might have. Figure 2 shows the classes with the changes and figure 3 visualizes the hierarchy of their inheritance with a class diagram.

```
class Unit
{
	Vector3 position;
	bool alive;
}
class CombatUnit: Unit
{
	int maxHp;
	int currentHp;
	void TakeDamage() { }
}
class EnemyUnit : CombatUnit
{
	int attackPower;
	void Attack() { }
	void Move(Vector3 direction) { }
}
class SpecialEnemyUnit: EnemyUnit
{
	bool canFly;
	bool isRanged;
	int attackRange;
}
```

Figure 2 – OOP Class example 2.

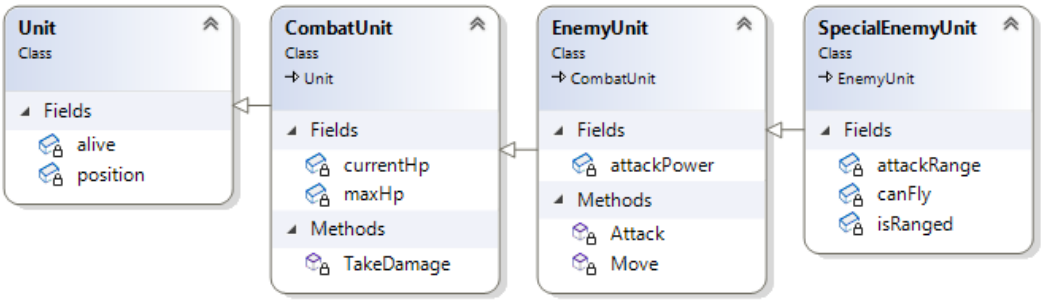

Figure 3 – Class diagram for OOP example in figure 2.

These inheritance problems could go on. For example, we might want a rolling stone unit that cannot be attacked but can move. We would need to use the `EnemyUnit` class even though it contains several useless variables and methods (i.e., alive, `maxHp`, `currentHp`, `TakeDamage`).

In OOP, the functionality and the data are tied together in the class. Thus, the class can act upon its data. In this way, OOP mimics the real world. For instance, if we create a dog class, we expect the dog to act independently and not need outside influence to function.

Continuing from the previous example, the `SpecialEnemyUnit` is 56 bytes in heap memory as exposed by Visual studio memory diagnostics (Figure 4).

Object Type	Count	Size (Bytes)
OOP_Examples2.SpecialEnemyUnit	1	56
OOP_Examples2.CombatUnit	1	48
OOP_Examples2.EnemyUnit	1	48
OOP_Examples2.Unit	1	40

Figure 4 – Sizes of classes in memory (Visual studio memory diagnostics).

A typical desktop processor's cache memory line (or block) is 64 bytes (Hennessy et al., 2017). If we want to change the `attackRange` of the `SpecialEnemyUnit` instance, we use 56 bytes of the 64-byte memory line (Figure 5). This is wasteful as only the int value that takes 4 bytes of memory was needed.

Figure 5 – 2 instances of `SpecialEnemyUnit` class visualized on a 64-byte memory line.

2.2.2 OOP problem solved using DOD

In OOP, each object is self-contained and can act on its data. In DOD, the data is not coupled with functionality. Therefore, it cannot act upon itself. This way, the data can be organized in groups and acted on in chunks (Figure 6).

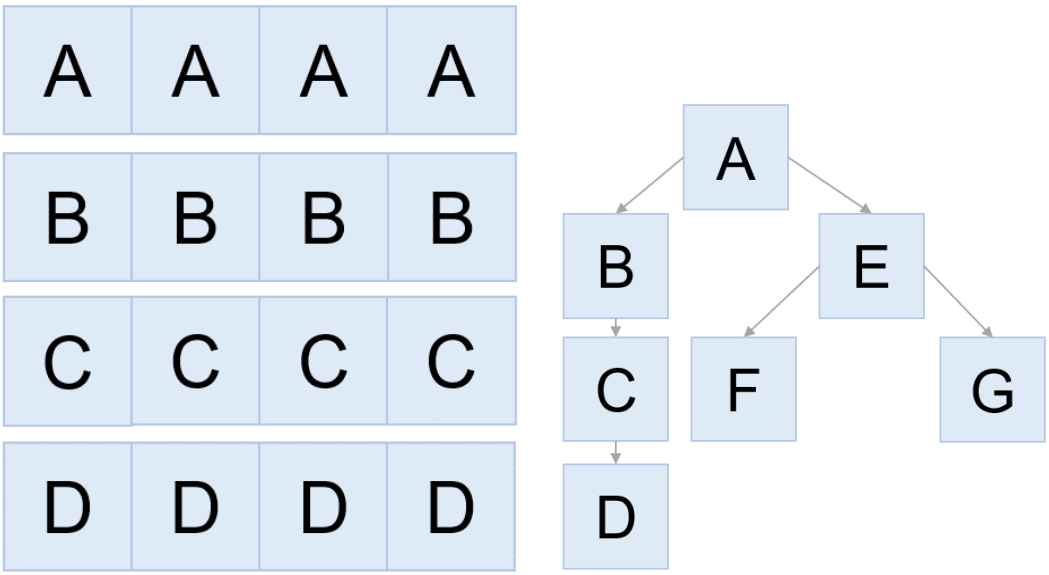

Figure 6 – DOD data on the left is organized sequentially in groups. The OOP data on the right is in a hierarchy tree with dependencies to its class inheritance.

From the previous OOP example, if we wanted to transform the `attackRange` value with DOD, the range would be made into a separate data structure that can then be transformed in chunks. The code has become more cache-friendly as we only use the data needed for the transformation (Figure 7).

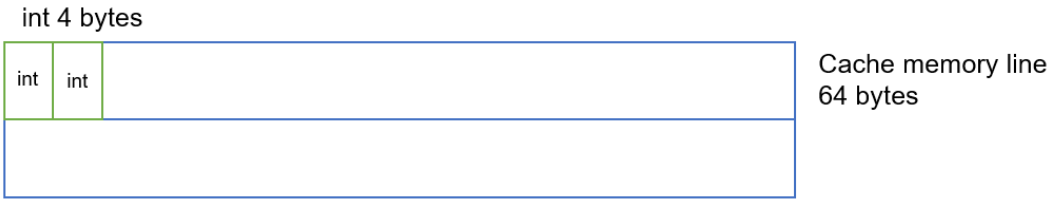

Figure 7 – two instances of `int` visualized on a 64-byte memory line.

With OOP, parallelization can be cumbersome as each object can act upon itself and might affect the other objects simultaneously. With DOD, the data is processed sequentially in groups. The grouping means that the order of the processes is known, and parallelization becomes simpler than ungrouped data.

This chapter has explained what the motives for using data-oriented design are. The next chapter discusses Unity’s approach to data-oriented design.

3 UNITY DOTS

Unity is a real-time 3D development platform developed by Unity technologies. In this work, the focus is on Unity's capabilities as a game engine. DOTS is Unity's approach to data-oriented design for game development in Unity (Unity 2021b). DOTS aims to utilize multithreading and cache optimization better than the current object-oriented approach. Unity also indicates that with their Data-oriented design, reusing code would be more straightforward.

DOTS is still in preview at the time of this project and has not been declared production-ready. This chapter explains the components of DOTS that are relevant to this work. DOTS also includes NetCode and DSPGraph, but they were not used or explored in this work. The NetCode preview package is used for DOTS net code and was not utilized as networks were not a part of the experiments. DSPGraph experimental package allows for audio systems to be used with DOTS. It was not used, as audio was not needed for the experiments.

3.1 Entity Component System

Entity Component System (ECS) is the core of Unity DOTS (Unity 2020c). As the name indicates, it has three primary elements:

- **Entities** replace game objects of the conventional Unity object-oriented system. They are the general things that populate the game world and have game logic.
- **Components** hold the data of the entities. They are organized separately from their entities. This is one of the major factors differentiating the data-oriented approach from the object-oriented one.
- **Systems** handle the game logic. In the object-oriented approach, it is typical to let each object handle its logic. However, with the data-oriented method, one system can handle the game logic of all the queried entities. For instance, a system could handle the movement logic of all the entities with the `MoverTag` data component.

Usage of these elements is what distinguishes the usual Unity object-oriented approach from DOTS. Some essential practical differences that they bring in development include:

- **Entities** and **components** cannot be changed in the inspector in runtime like conventional `MonoBehaviour` game objects.
- **Entities** need to be created in the code and not manually in the inspector. Entities are usually created in runtime rather than while editing the scene in the editor. Though, the conversion workflow approach makes this more intuitive.
- **Systems** are not `MonoBehaviour` type game objects in the world. Unity automatically finds and instantiates **systems** from the project.
- **Systems** control game logic with `ForEach` (or job chunk) entity functionality. These require queries to get the wanted data **components** and **entities** to manage. Even if there is only one **entity** to be controlled in a system, it is done using these functions.

3.1.1 Conversion Workflow

The conversion workflow converts `GameObjects` into entities (Unity 2021d). The `GameObjects` are given authoring components that the conversion system then converts into data components. The `GameObject` itself is converted into the entity that holds these components. This workflow allows a more friendly approach for developers coming from conventional Unity object-oriented programming than pure code-based component creation.

The following example clarifies the steps of authoring components for entities. A developer creates a primitive cube type `GameObject` and checks the `ConvertToEntity` box in the inspector. The cube is then converted to an entity when the game starts. The new entity looks identical to the game object that it was converted from. However, as the cube entity does not connect to the original object, the original cube can safely be destroyed when the entity is created. See Figure 8 for the check box and conversion mode that destroys the object after conversion.

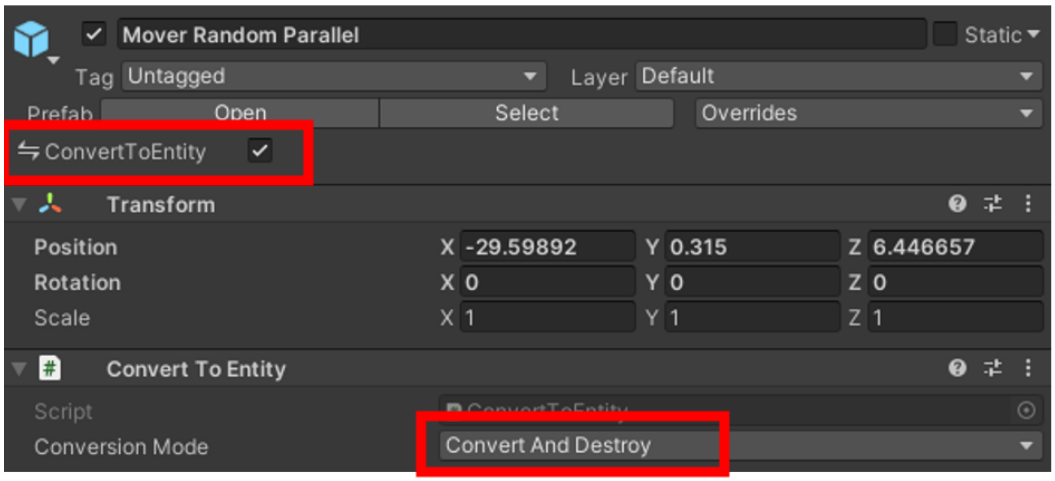

Figure 8 – Convert to entity check box and conversion mode.

Authoring components, which are `MonoBehaviour` scripts, could also be added to the `GameObject`, which would convert into data components. Authoring components can expose values in the inspector that will be converted to values for the data component (Figure 9). Systems can then query for these data components to add game logic to the entity and components (Figure 10).

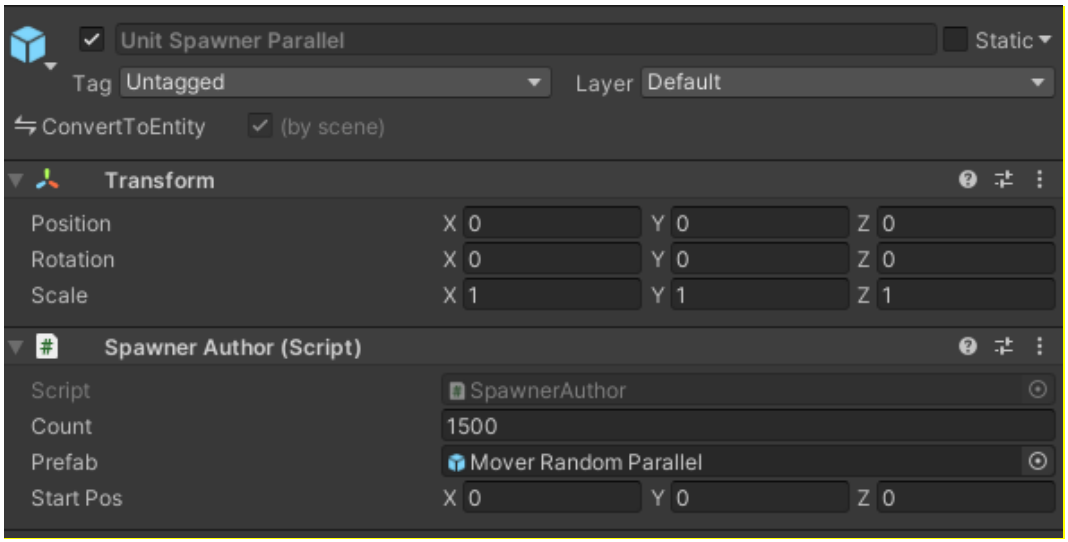

Figure 9 – Spawner Author authoring component has values that can be changed in the inspector.

```
Entities.ForEach((Entity entity, ref EnemyData data, ref Translation translation) =>
{
	data.value = translation.Value.x;
}).ScheduleParallel();
```

Figure 10 – Entity `ForEach` Query looks for entities with `EnemyData` and `Translation` components.

Unity (2021d) has stated that the conversion workflow is not temporary and is a fundamental part of ECS development. They also advocate that it is the preferred way of authoring data for entities when using ECS.

3.1.2 Command Buffers & Update Groups

`EntityManager` can make structural changes to entities. These changes include adding or removing components from entities. However, `EntityManager` cannot be accessed inside jobs (Unity 2021d). Instead, ECBs (entity command buffer) are used to create structural changes at set sync points after the job is complete.

For instance, component `MoverTag` needs to be removed from the entity after its movement has been calculated. However, the component cannot be removed while the movement calculation job is running. So instead, the component removal is queued to an ECB which executes the removal after the job is complete.

The sync points for the ECBs are mainly used in combination with the update groups. The update groups create sync points where multiple ECBs can execute in the order of their creation. However, too many sync points can slow down the code, as jobs need to wait for their execution to ensure safety. With update groups, the sync points are consistent and easy to keep in check. The most prominent update groups used in this work include the following groups:

- `InitializationSystemGroup` - updated at the end of the Initialization phase of the player loop.

- `SimulationSystemGroup` - updated at the end of the Update phase of the player loop.
- `FixedStepSimulationSystemGroup` – updated according to the fixed-step interval of the simulation group. Same usage case as `FixedUpdate` in `MonoBehaviour` scripts.

Using ECBs and update groups is a major part of the effective usage of ECS. Though, structural changes and sync points can have a considerable negative performance impact if overused.

3.1.3 Dynamic buffers

A dynamic buffer is a type of data component an entity can have. It can hold a number of variables like an array. The buffer data can be managed inside jobs (Unity 2021d). Dynamic buffers behave similarly to other entity data components and can be referenced in an entity query. This gives them a similar usage case to arrays and lists in C#.

The memory allocation of dynamic buffers is managed automatically by the entity manager. The internal capacity of the buffers can be declared in code by the developer. If the internal capacity of the buffer exceeds the set capacity, the data is moved onto the heap by the entity manager. Dynamic buffers are used in the pathfinding implementation to maintain path data for the moving units.

3.2 Hybrid renderer

Hybrid renderer handles date for rendering entities of ECS (Unity 2020a). It is not a rendering pipeline like URP (Universal Render Pipeline) or HDRP (High-Definition Render Pipeline). Instead, it gathers necessary entity data to send to the actual pipelines. The hybrid renderer does not require any additional changes or steps to use when creating entities. The developer only needs to install the package, and the renderer works in the background.

Two versions (i.e., V1, V2) of the hybrid renderer are available. V1 renderer supports the built-in pipeline in addition to URP and HDRP. V2 Supports only

URP and HDRP but has many features not available in V1 (e.g., Shader graph, Skinning, Sunlight, Lightmaps). V2 was used for the implementation as shader graphs and skinning were needed for the animations.

3.3 C# Job system

The job system allows the writing of multithreaded C# code (Unity 2021b). It exposes Unity's internal C++ job system, allowing C# scripts to run as jobs parallel to Unity components. Jobs are small workloads with specific tasks that can wait for other jobs to complete or run parallel to them. Multithreaded code can provide high-performance benefits in specific tasks.

The following example showcases how multithreading could be utilized. A CPU has eight cores and eight worker threads to work with. The game has a single character controller and multiple enemy AIs. Typically, in Unity, all game logic would be called from the main thread. However, we could offload some of this from the main thread with the job system and create jobs for the enemy AI. These jobs could then run parallel on the worker threads to alleviate the stress on the main thread. As a result, the code became easier to run and more efficient (Unity 2021b). This is advantageous, especially if factors like battery saving on mobile devices are taken into consideration.

3.4 Burst Compiler

Machine code is a numerical (e.g., Binary, Hexadecimal) programming language that a CPU can understand directly. CPUs have varied architectures (e.g., x86-64, ARM 64) with their native machine code. Optimizing machine code has considerable potential for performance gains as it is the direct way to control the CPU.

The burst compiler translates .NET bytecode into highly-optimized native machine code using LLVM (not an acronym) (Unity 2021b). LLVM is a library for programmatically creating machine native code (Yegulalp 2020). IL and IR are

intermediates that .NET applications can create to be translated into machine code. Burst translates C# code from IL (Intermediate Language) to IR (Intermediate Representation) (Mutel 2018). LLVM handles the heavy lifting, so to say and compiles the IR to native machine code.

Burst uses a subset of .NET that Unity calls HPC# (High-Performance C#) (Mutel 2018). This subset removes most usage of managed types. It supports the following primitive types of C# (Unity 2020d):

- `bool`
- `sbyte/byte`
- `short/ushort`
- `int/uint`
- `long/ulong`
- `float`
- `double`

The commonly used type `string` is not supported as it is a managed type. However, Unity (2021f) informs that future releases will support the `char` type.

Unity Mathematics is a C# library that provides burst with vector types and math functions. Burst translates Unity Mathematics' vector types to native SIMD (Single Instruction Multiple Data) vector types. For example, the types `float3` and `int3` are used in place of `Vector3`.

Burst requires the usage of structs instead of classes. Structs are a value type stored on the stack and do not need to be managed. Classes are a reference type that needs to be managed on the heap, making them incompatible with burst. These limitations are one of the major differences to conventional object-oriented Unity programming.

3.5 Unity Physics

Unity Physics package, not to be confused with the built-in physics in Unity, is a physics solution built to work with DOTS (Unity 2021c). The package was created in collaboration with Havok, who are behind the Havok Physics engine. The build-in physics is not compatible with DOTS, so the Unity physics package (or Havok Physics for Unity) must be used for physics with DOTS. Unity Physics package covers many of the same functions as the build-in physics (e.g., Collision, Gravity, Ray casting, Collider cast, Point Distance, Overlap Queries).

Unity Physics is stateless (Unity 2021c). Statelessness implies that it does not cache data, which other physics engines use to attain high performance and simulation robustness. Caching can complicate some use cases, such as networking. Unity Physics leaves caching out to be simpler and more controllable for the developer.

3.6 Unity Animation

The Unity Animation package is Unity's solution for DOTS compatible animations (Unity 2021b). While it is still experimental and lacks many features, it supports the essential animation features (i.e., animation blending, IK, root motion, layers, and masking). There is not much info on the package's inner workings or design philosophy from Unity, which is expected from an experimental package. It is not production-ready, and only the core features are currently supported.

3.7 DOTS relevant C#

DOTS makes use of some C# code that is not usually seen in object-oriented Unity code. The reason is that they are simply not needed in it. This part describes and explains the reasons behind these features.

3.7.1 Parameter Modifiers

There are two parameter modifier keywords (i.e., `ref`, `in`) that get considerable usage in DOTS. The keyword `ref` passes variable by reference with read and write access. The keyword `in` also passes the variable by reference but with only read access.

Structs are used extensively in DOTS in place of classes. Structs are value types and usable inside jobs. The `ref` keyword is used when a struct needs to be referenced because assigning a struct variable creates a copy rather than a reference (Microsoft 2021a). For example, querying for an entity `ForEach` job requires the usage of the `ref` or `in` keywords.

The `out` keyword passes variable by reference with only write access, which can return multiple values from a method. The `out` keyword is helpful for DOTS applications but not as relevant as `ref` and `in`, as entity `ForEach` jobs do not require the usage of `out`.

3.7.2 Unsafe C# code

Generally, when writing C# code, managed objects are created (Microsoft 2021b). With managed types, memory is not directly accessed or allocated, making code verifiably safe, meaning the .NET tools can verify the safety of the code. Unsafe code means unverifiable code. Unsafe code is not inherently dangerous; it just is not verifiably by the .NET tools. With the unsafe keyword in C#, a developer can access pointers, allocation of memory blocks and the calling of methods using function pointers.

3.7.3 Pointers in DOTS

Pointer is a variable that holds the address of a variable in memory (Microsoft 2021). In other words, the pointer's value is the address and not the value of the variable itself. Pointers are helpful in DOTS because memory blocks used for

the pointer variables can be manually allocated, making them unmanaged by the garbage collector, allowing them to be used inside jobs. The implementation of the DOTS pathfinding showcases one example of pointers in use.

4 RELATED WORK AND HYPOTHESIS

A decent amount of recent work on DOTS has been done (Männistö 2020; Borufka 2020; Turpeinen 2020; Codemonkey 2020; Bad Graphix 2020). However, not a great amount, as its packages are still in preview or experimental. Furthermore, Unity has been quiet on DOTS updates since the end of 2020, which most likely has not been encouraging for anyone trying to get into the topic.

This chapter compiles some recent and relevant work on DOTS. It also discusses the research questions and hypotheses relating to them.

4.1 DOTS in production

Männistö (2020) created two game demos. The first demo used conventional Unity tools, and the second applied DOTS tools. The test aimed to research the difference between planning and creating these demos. Both approaches spawned enemy units until 20 000 units were reached. Männistö used CPU response times as their measurement for the performance test. The response time was noted at set amounts of units spawned (i.e., 1000, 5000, 10 000, 15 000 and 20 000) to observe the difference the amount of the units make to the CPU response time.

The test results were that DOTS performed up to 3 times better in the test scenario. Männistö considered that the performance of DOTS was promising but did not recommend DOTS to be used for production. Männistö identified several problems that would limit its applicability in game development. The major problems were the lack of documentation from Unity and the complex code syntax. Männistö concludes that more testing needs to be done, especially with any newer version of DOTS.

4.2 Job Scheduling & burst in DOTS

Multithreading code with the job system is a major part when creating DOTS systems. When using the recommended `SystemBase` extension for systems in DOTS, there are three ways to schedule a job; parallel (i.e., `ScheduleParallel`), single (i.e., `Schedule`) and the main thread (i.e., `Run`) (Unity 2020b). Parallel schedules the job for concurrent execution on available worker threads. Single schedules the job on a single available worker thread, but not concurrently with other jobs. The main thread performs the job on the main thread immediately. DOTS can also enable burst compilation. Burst limits the data types that can be used and, in return, produces highly optimized machine code.

Borufka (2020) tested the performance of DOTS relevant datatypes and settings. They created a performance test suite that could run different types of tests with given parameters. One of the tests sets Borufka ran included the Job Type test set. They compared six different job types (i.e., No Job, Job, Parallel Job, Burst Job, Burst Parallel Job, Burst 5 Jobs). The results of the test were that burst compiled jobs were faster than their non-burst counterparts. Based on the tests results, Borufka concluded that single job scheduling can be faster than parallel scheduling in smaller datasets of 1000 and below. However, the parallel jobs were the fastest in the larger datasets of 10 000 and above.

4.3 Visualization of performance tests

Since Borufka’s (2020) tests were not visualized inside Unity, it was also concluded that it would be beneficial to create benchmarks that visualize the data directly inside the Unity Editor or a standalone build. This would decrease the time to analyze and visualize the data.

4.4 Pathfinding

Code Monkey (2020) implemented A DOTS pathfinding system using the grid-based A* algorithm with DOTS (Code Monkey 2020). A* star is popular as it tries to find the most promising path to the target location, skipping over some unpromising tiles in the process (Hybesis 2020). This makes it useful for videogames when efficient pathfinding for an AI is needed. The A* algorithm has been compared to other pathfinding algorithms in several studies and was concluded to be more efficient in comparison (Zeng et al., 2009; Permana et al., 2018). Furthermore, a DOTS Flocking system using the Boids algorithm has been implemented (Bad Graphix 2020).

These references (Code Monkey 2020; Bad Graphix 2020) prove that creating pathfinding applications with DOTS is possible. However, they did not perform systematic analysis on their testing, giving a reason for a move systematic approach on the topic.

4.5 Virtual Reality

Framerate is vital to get the best experience in virtual reality and achieving the target framerate for new high refresh-rate VR headsets is not trivial. VR requires every frame to be drawn twice for each eye (Oculus 2021) which is the primary reason that makes VR heavy to run, and thus it requires powerful hardware and excellent optimization from the developer.

Tcha-Tokey et al. (2017) tested five factors (i.e., interaction level, framerate, field of view, 3D content feedback, previous experience) influencing subjective user experience components and objective usability. They had 152 participants in their testing. The tests were run in VR applications specifically. On average, the testers rated higher framerate for better usability. Therefore, they concluded that VR applications should use higher framerates for better usability. Which, in turn, enforces the pursue to optimize VR applications for higher framerates.

4.6 Mobile Augmented Reality

Turpeinen (2020) ran tests on two iPhone models. These tests measured the difference in the performance of object-oriented (OO) and DOTS approaches on a mobile platform. Turpeinen spawned the same number of units using both methods and compared CPU response times.

In the 10% original mesh test on the iPhone XR, at 1000 characters, DOTS held the 30 FPS target frame rate while OO was at 19 FPS. They also mention that OO can perform better with small amounts of units. In their animation testing, the OO approach was better with the variance of animation, which meant that the OO performance would be more consistent as the animations of the characters became more varied. Turpeinen concludes that DOTS performance benefits become more apparent as the number of units increases.

4.7 Research questions

Based on the previous discussion in this chapter, the following research questions were gathered:

1. Is the current version of DOTS production-ready?
2. Can DOTS performance data be visualized inside Unity Editor?
3. What scheduling type is the fastest in visualized performance tests?
4. Can complex simulation pathfinding be created in DOTS, and how well will it perform?
5. Is DOTS compatible with VR, and how well does it perform?
6. Is DOTS compatible with mobile AR, and well does it perform?

4.7.1 Is the current version of DOTS production-ready?

Männistö (2020) did not conclude DOTS to be production-ready as the documentation on DOTS was lacking at the time.

From the time of Männistö's work, Unity's DOTS documentation and example projects have been updated, making creating DOTS applications more

accessible in general. The prediction on the production-readiness of DOTS is that since Unity has not yet declared it production-ready, the research is unlikely to change that fact. However, if the performance gains are significant enough compared to the conventional object-oriented Unity, DOTS could be declared production-ready in certain usage cases.

4.7.2 Can DOTS performance data be visualized inside Unity Editor?

Borufka (2020) concluded that performance tests with visualized data inside Unity should be run. Their performance tests were comprehensive but did not include visuals.

There should be no issues with the visualization of data inside the editor. Rendering units is not anticipated to be an issue because Unity has created the hybrid renderer for this task. Additionally, DOTS should be compatible with Unity's UI components as previous DOTS projects have used it (Codemonkey 2020).

4.7.3 Can complex simulation pathfinding be created in DOTS, and how well does it perform?

Pathfinding and flocking have been a topic of research for a long time (Dijkstra 1959; Reynolds 1987), and as such, they have been researched extensively (Zeng et al. 2009; Permana et al. 2018; Krishnaswamy 2009). Though work done for Unity DOTS pathfinding explicitly is limited, it would be worthwhile to have more data on this.

The hypothesis for the pathfinding tests is that DOTS pathfinding should be more efficient than conventional Unity tools, in this case, the non-experimental navigation tools in Unity. The prediction comes from Unity's promise for performance and previous work showing promising results (Turpeinen 2020). The tools used for this will be discussed more in-depth in the implementation.

4.7.4 Is DOTS compatible with VR, and how well does it perform?

Tcha-Tokey et al. (2017) concluded that a higher framerate contributes to better objective usability in VR applications. One way to optimize for a greater framerate is to limit time spent in scripts (Oculus 2021).

The hypothesis is that using DOTS in Unity will significantly reduce time spent in scripts compared to similar operations done with conventional object-oriented Unity tools. In addition, there should be no combability issues with DOTS and VR because VR should not interfere with the code. However, the rendering might have issues as DOTS uses the hybrid renderer, and it is not clear if the hybrid renderer supports VR rendering.

4.7.5 Is DOTS compatible with mobile AR, and how well does it perform?

Turpeinen (2020) displayed that DOTS is compatible with mobile platforms, but DOTS combined with AR has limited research.

The prediction is that there should not be any compilations because the AR and the DOTS systems should work separately. This research explores any unwanted interactions AR and DOTS might have.

4.8 Conclusion of hypothesis

This chapter highlighted some research questions from previous work on the topic of DOTS. The main research question of the work is the following: Is the current version of DOTS production-ready? The other questions offer research topics that are used to get a better understanding of this. VR and AR compatibility are the secondary topic of this research and are researched using similar methods.

The author did not have previous experience with Data-oriented design before conducting this research but had experience with object-oriented Unity

development. This experience difference might skew the results towards DOTS not being production-ready as more experience gained with DOTS might make it more usable for the developer.

5 METHODS

The primary method of testing was pathfinding using DOTS and conventional object-oriented Unity tools. Pathfinding was chosen as it has many applications in games. A common one is the movement of units from place A to B. Moving units are also a good way to visualize the tests.

The DOTS pathfinding was run as a benchmark to compare scheduling types and the DOTS approach to the object-oriented approach. In addition, animations with both approaches were run to compare the impact of blending animations on their performance. Finally, the same pathfinding was applied in AR and VR to determine compatibility and performance.

5.1 Pathfinding benchmark environment

The pathfinding benchmark environment spawns new units until the FPS hits under 10 FPS. These units find paths to randomized destinations in the set area. 1250 Units are spawned every 0.1 seconds for most of the benchmarks. The Parallel Animated and Mono Benchmarks had different spawning cycles as they could not handle this number of spawning units without dipping to under 10 FPS in the first spawns. The time to reach 10 FPS was the primary measurement for the benchmarks. More time spent in the benchmark equals better results.

As the Parallel Animated and Mono benchmarks had different spawning cycles, the number of units spawned was measured for them instead. In addition, all the benchmarks in this environment were run in the Unity Editor because the standalone build did not support the `BlopAssetStore` code used for the animations.

Each benchmark was run multiple times during development to ensure consistency. Final results were taken from an average of 3 benchmark runs on each device.

The following nine tests were run in the pathfinding benchmark environment:

1. **Parallel Burst** – Pathfinding is scheduled in parallel with burst enabled.
2. **Parallel No Burst** – Pathfinding is scheduled in parallel with burst disabled.
3. **Single Burst** – Pathfinding is scheduled on single worker threads with burst enabled.
4. **Single No Burst** – Pathfinding is scheduled on single worker threads with burst disabled.
5. **Run Burst** – Pathfinding runs on the main thread with burst enabled.
6. **Run No Burst** – Pathfinding runs on the main thread with burst disabled.
7. **Parallel Animated** – Pathfinding is scheduled in parallel with burst enabled. Spawned units have blending animations (10 units spawned every 0.25 seconds).
8. **Mono** – Pathfinding using conventional object-oriented Unity tools (500 units spawned every 0.5 seconds).
9. **Mono Animated** – Same as mono with the addition of animations for units.

The first seven benchmarks represent DOTS approaches, and the last two the object-oriented approaches. The results of the first six benchmarks were used to compare the scheduling types and burst. The best result of the first six was compared to the Mono pathfinding to measure pathfinding performance difference. Finally, parallel animated and Mono animated were compared to measure which approach performed better with animations.

Table 2 – Specifications of PCs benchmarked in the pathfinding environment.

PC	**Desktop**	**Laptop**
OS	Windows 10 Home	Windows 10 Home
CPU	AMD Ryzen 7 3800X 8-core (16 threads) 3.9 GHz (Boost 4.5 GHz)	Intel Core i5-7300HQ 4-core (4 threads) 2.5 GHz (Boost 3.5 GHz)
GPU	NVIDIA GeForce RTX 3070 8GB GDDR6	NVIDIA GeForce GTX 1050 2GB GDDR5
RAM	16 GB DDR4 3200 MHz (Dual Channel)	8 GB DDR4 2400 MHz (Single Channel)

The benchmarks were run on two PCs (Table 2). Hypothetically the desktop should outperform the laptop in all benchmarks because the desktop's components are much newer and more performant. The main factor for the performance should be the CPU. Using the job system allows for the usage of all the threads of the CPU. Therefore, a processor with good multithreaded performance should perform well in the benchmark, compared to a CPU with fewer threads but better single-threaded performance. Unfortunately, such a less threaded CPU was not available within the limitations of the work.

Figure 11 shows the benchmarking environment with options for the different benchmarks on the right side. The measurement for the unit count and elapsed time can be found above the buttons for running the benchmarks. The left side includes the specifications of the current device and FPS metrics. The specifications include from top to bottom: Orange for GPU and VRAM, blue for CPU and RAM and yellow for the operating system. Below the specifications, in the FPS metrics, green is current FPS, orange FPS lows, blue FPS highs and

yellow FPS change range.

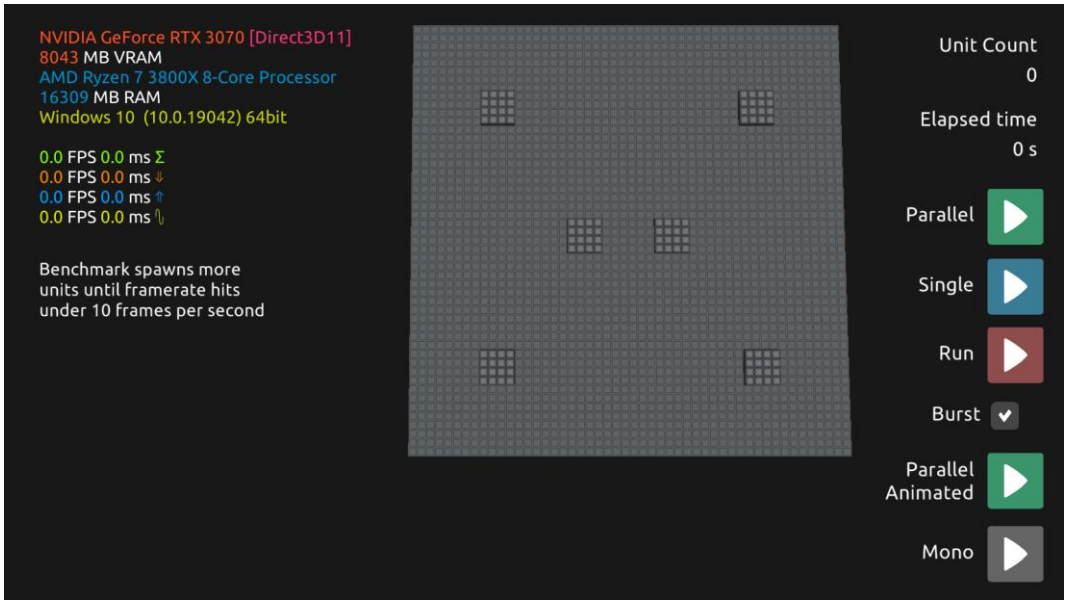

Figure 11 – Pathfinding benchmark environment.

5.2 VR environment

The VR tests were run on the same desktop as the pathfinding. VR headset used for the tests was the Oculus Rift S. VR was tested in both Unity editor and standalone build. The VR environment was built as a tower defence type game (Figure 12) to test the hybrid development approach with DOTS and Steam VR Unity packages.

The enemy units use the same pathfinding as in the pathfinding environment. Instead of randomized destinations, the tower defence enemy units have a set destination. Enemies spawn from 2 different spawns and have the same set destination for pathfinding. The towers try to find enemies in their range to attack and destroy. The player can place towers and obstacles to destroy the enemies more efficiently. No resource system was implemented in this environment, which meant the environment was practically a sandbox with no challenge.

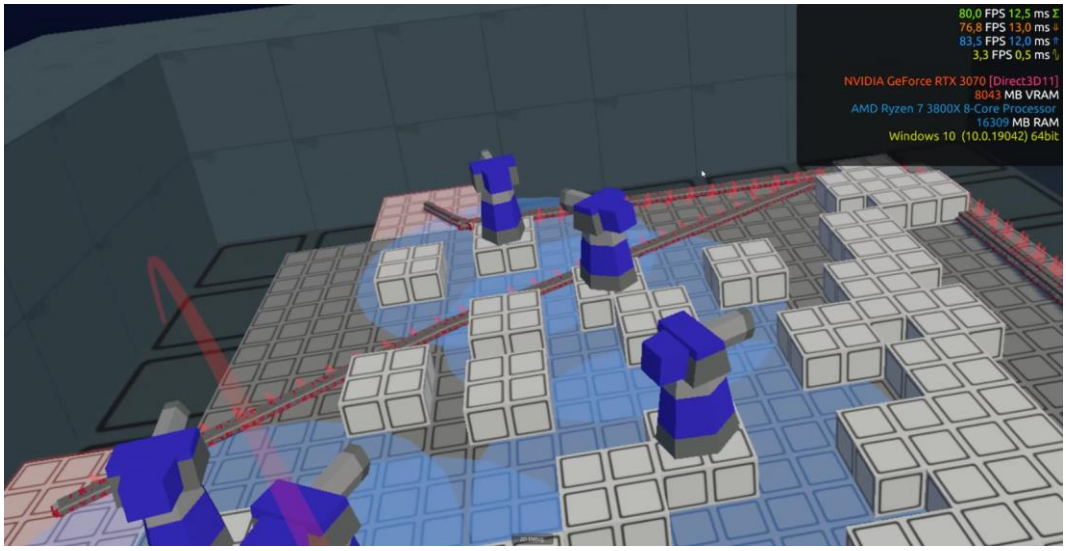

Figure 12 – VR testing environment with tower defence mechanics.

5.3 AR environment

In the AR environment (Figure 13), a plane was detected in the real world. When the plane was of a suitable size, a navigation mesh was baked on it. Units were spawned in intervals of 500 per spawn. Units have the same pathfinding as in the pathfinding environment. More units were spawned, and the framerate was observed. The tests were run until 20 000 units had been spawned.

Figure 13 – AR testing environment with units spawning on an AR plane.

Table 3 – Specification of phones used in the AR environment.

Phone	**Google Pixel 3A**	**LG Nexus 5X**
OS	Android 12 (beta 4)	Android 8
CPU	Qualcomm Snapdragon 670 8-core	Qualcomm Snapdragon 808 6-core

The tests were run on two devices (Table 3) to understand the performance difference between an older and a newer device. The Pixel is an average phone by 2021 standards (PassMark Software 2021b), making it a worthwhile candidate to test if DOTS is worthwhile to use in AR on mobile. The Nexus was released in 2015, making it an obsolete phone compared to new models. However, if DOTS performs well, it could be important for how DOTS could be utilized for older devices. Both phones are from the same Google line of

phones, with the Pixel being a newer model. The Pixel should outperform the Nexus as it has a more powerful processor with more cores.

5.4 Pathfinding implementation

The most prominent part of the implementation was pathfinding. The implementation of the VR and AR environments will also be discussed. The versions of Unity and the packages are found in Table 4. Unity version 2020.3 was required as DOTS was not supported on 2021 versions at the time.

Table 4 – Unity and package versions.

Editor/Package	**Version**
Unity Editor	2020.3.10f1
Entities	0.17.0-preview.42
Hybrid Renderer	0.11.0-preview.44
Burst	1.4.1
Jobs	0.8.0-preview.23
Mathematics	1.2.1
Physics	0.6.0-preview.3
Animation	0.9.0-preview.6
Steam VR	2.7.3
AR Foundation	4.1.7

5.4.1 Pathfinding using experimental AI components

The chosen system for the pathfinding was Unity's experimental AI components package because it is compatible with the job system (Unity 2021e). The non-experimental Unity AI is not compatible with the job system. The main components of it are `NavMeshLocation`, `NavMeshQuery` and `NavMeshWorld`. The experimental AI was chosen as the research on its performance is limited,

and other algorithms such as A*, Dijkstra and D* have already been proven to be efficient (Krishnaswamy 2009).

In addition to the Experimental AI, the `NavMesh` component package by Unity was used. It allows baking the navigation mesh in runtime with the `NavMeshSurface` component. This was important for the AR because the `NavMesh` had to be baked in runtime according to the found AR plane in the real world. This will be discussed in more detail later in the section on the AR implementation. `NavUtils` script by Unity is used to find the straight path after the pathfinding is ready. It was created for Unity's Austin 2017 demo (Unity 2017).

Several references for the usage of `NavMeshQuery` have been taken from Reese's DOTS Navigation (Schultz 2021), which is an extensive Unity package for DOTS based navigation.

`NavMeshQuery` is used to query data from the `NavMeshWorld`. `NavMeshWorld` contains the needed data to create pathfinding, such as the current available `NavMeshSurfaces`. It can operate inside the job system, which the normal `NavMesh` operations cannot, making it crucial for best performance results with DOTS.

For it to be accessible through parallel jobs, the queries must be accessed through pointers because creating new queries is not allowed while running a parallel job. This is one limitation of the experimental package, resulting in unsafe code needed to accomplish the desired result (Figure 14).

```
unsafe public class NavQueryPointerSystem : SystemBase
{
    unsafe public struct NavMeshQueryPointer
    {
        [NativeDisableUnsafePtrRestriction]
        public void* query;
    }

    public NativeArray<NavMeshQueryPointer> QueryPointers { get; private set; }
    List<NavMeshQuery> queryList = new List<NavMeshQuery>();

    protected override void OnCreate()
    {
        var queryPointers = new NavMeshQueryPointer[JobsUtility.MaxJobThreadCount];
        for (int i = 0; i < queryPointers.Length; i++)
        {
            queryPointers[i] = new NavMeshQueryPointer
            {
                query = UnsafeUtility.Malloc(
                        UnsafeUtility.SizeOf<NavMeshQuery>(),
                        UnsafeUtility.AlignOf<NavMeshQuery>(),
                        Allocator.Persistent
                    )
            };
            var query = new NavMeshQuery(
                    NavMeshWorld.GetDefaultWorld(),
                    Allocator.Persistent,
                    NavConst.PathNodePoolSize
            );
            queryList.Add(query);
            UnsafeUtility.CopyStructureToPtr(ref query, queryPointers[i].query);
        }

        QueryPointers = new NativeArray<NavMeshQueryPointer>(queryPointers,
Allocator.Persistent);
    }
    …
}
```

Figure 14 – Script (Schultz 2021) for creating pointers that can be used to access the queries.

`NavQueryPointerSystem` creates a query for each available worker thread, as that is the maximum number of queries that can be used in parallel at any given time. These pointers can then be used whenever a `NavMeshQuery` operation needs to be executed. As they are pointers, they need to be manually disposed of when the application is stopped.

The destination must first be confirmed to be on the `NavMesh` to find a path using `NavMeshQuery`. Points not on the `NavMesh` are not valid for pathfinding. To do this, `NavMeshQuery` methods `IsValid` and `MapLocation` are used. `MapLocation` finds the closest point in the `NavMesh` given a maximum search

extent. `IsValid` checks if the point is active on the NavMesh. The code in Figure 15 demonstrates an example of this operation.

```
NavMeshLocation location = query.MapLocation(searchPosition, searchExtent,
areaMask);

if (query.IsValid(location))
{
    foundLocation = location;
    return true;
}
```

Figure 15 – Code validates the map location.

When the destination location has been successfully validated, finding a path to it can be started. The start and end locations need to be inputted into the `BeginFindPath` method to find the path. The path is then iterated as many times as is needed for the wanted accuracy. After the path has been iterated on, the pathfinding can be ended. Finally, the `FindStraightPath` finds the straightest path to the destination. These parts of the code have been highlighted in Figure 16.

```
var status = query.BeginFindPath(startLocation, endLocation);

while (status == PathQueryStatus.InProgress)
    status = query.UpdateFindPath(updateIterations, out _);


if (status == PathQueryStatus.Success)
{
    query.EndFindPath(out int pathSize);
    int pathMax = pathSize + 1;

    path = new NativeArray<NavMeshLocation>(pathMax, Allocator.Temp);
    int straightPathCount = 0;
    NativeArray<float> vertexSide = new NativeArray<float>(pathMax, Allocator.Temp);
    NativeArray<StraightPathFlags> straightPathFlags =
    new NativeArray<StraightPathFlags>(pathMax, Allocator.Temp);

    NativeArray<PolygonId> polgygonPath =
    new NativeArray<PolygonId>(pathSize, Allocator.Temp);
    query.GetPathResult(polgygonPath);

    PathQueryStatus straightStatus = PathUtils.FindStraightPath(
        query,
        startLocation.position,
        endLocation.position,
        polgygonPath,
        pathSize,
        ref path,
        ref straightPathFlags,
        ref vertexSide,
        ref straightPathCount,
        pathMax
    );
}
```

Figure 16 – Code used for finding the straight path from the start location to the end location.

5.4.2 Destination & movement systems

To utilize the created pathfinding, a system to give destinations for the units had to be created. First, the system finds a random valid point on the `NavMesh`. With the valid point found, pathfinding can be started. If it is successful, the system gives the path to the dynamic buffer component of the entity. The movement system then handles the movement of the entity. These steps are highlighted in Figure 17.

```
[UpdateInGroup(typeof(SimulationSystemGroup))]
public class NavQueryRandomParallelSystem : SystemBase
{
    protected unsafe override void OnUpdate()
    {
        var surfaceSystem = World.GetExistingSystem<NavSurfaceSystem>();

        if (surfaceSystem.surface == null)
            return;

        var surfaceExtent = surfaceSystem.surfaceExtent;
        var surfaceCenter = surfaceSystem.surfaceCenter;

        var navMeshQueryPointers = World.GetExistingSystem<NavQueryPointerSystem>().QueryPointers

        Entities
        .WithNativeDisableParallelForRestriction(navMeshQueryPointers)
        .WithAll<NavAgentRandomTagParallel>()
        .ForEach((Entity entity, int entityInQueryIndex, int nativeThreadIndex,
        ref DynamicBuffer<QueryPosition> positionBuffer, ref SeedIndex seedIndex,
        in Translation translation) =>
        {
            if (positionBuffer.Length == 0)
            {
                var navMeshQueryPointer = navMeshQueryPointers[nativeThreadIndex];
                UnsafeUtility.CopyPtrToStructure(navMeshQueryPointer.query,
                    out NavMeshQuery query);

                if (NavUtility.RandomLocationOnSurface(surfaceExtent, surfaceCenter,
                     NavConst.SearchExtent,(uint)entityInQueryIndex + seedIndex.index, query
                    out NavMeshLocation endLocation)
                 && NavUtility.FindPath(translation.Value,
                     endLocation.position, query,
                    out var path))
                {
                    positionBuffer.Clear();
                    for (int i = 0; i < path.Length; i++)
                     positionBuffer.Add(
                     new QueryPosition { position = path[i].position });
                    seedIndex.index++;
                }
                else
                    Debug.Log($"Invalid location at {entityInQueryIndex}");
            }
        }).ScheduleParallel();
    }
}
```

Figure 17 – Destination system for random destinations scheduled in parallel.

The movement system transforms the translation component of the entity to move it in the world. This movement does not have any collision checks and only follows the path given by the destination system. The movement system moves the entity until it is close enough to the destination point of the path. After this, the destination point is removed from the dynamic buffer component of the

entity. This continues until the buffer is empty and the destination system can give a new path for the entity. These steps are highlighted in Figure 18.

```
[UpdateInGroup(typeof(FixedStepSimulationSystemGroup))]
public class NavQueryMoveSystem : SystemBase
{
    protected override void OnUpdate()
    {
        Entities
        .WithAll<QueryMoverTag>()
        .ForEach((ref DynamicBuffer<QueryPosition> positionBuffer,
                  ref Translation translation) =>
        {
            // Sometimes mover can get a NaN position
            // Clear the buffer to try find a new position

       if (float.IsNaN(translation.Value.x) && positionBuffer.Length >= 1 )
            {
                translation.Value = positionBuffer[0].position;
                positionBuffer.Clear();
            }

       else if (positionBuffer.Length > 1 && !float.IsNaN(translation.Value.x))
            {
                float speed = 10f;

                var currentPosition = positionBuffer[0].position;
                var destination = positionBuffer[1].position;
                float distance = math.distancesq(translation.Value, destination);
                // Stop movement when close enough to destination
                if (distance < 0.35f)
                {
                    positionBuffer.RemoveAt(0);
                }
                var direction = math.normalize(destination - currentPosition);
                translation.Value += direction * 0.02f * speed;
            }
            else if (positionBuffer.Length == 1)
                positionBuffer.Clear();
        }).ScheduleParallel();
    }
}
```

Figure 18 – Movement system script for units, scheduled in parallel.

The activity diagram in Figure 19 visualizes the workflow of the systems. The Destination system and the movement system work independently of each other. The destination system only cares about the entities that do not have a path and the movement system only about those with a path.

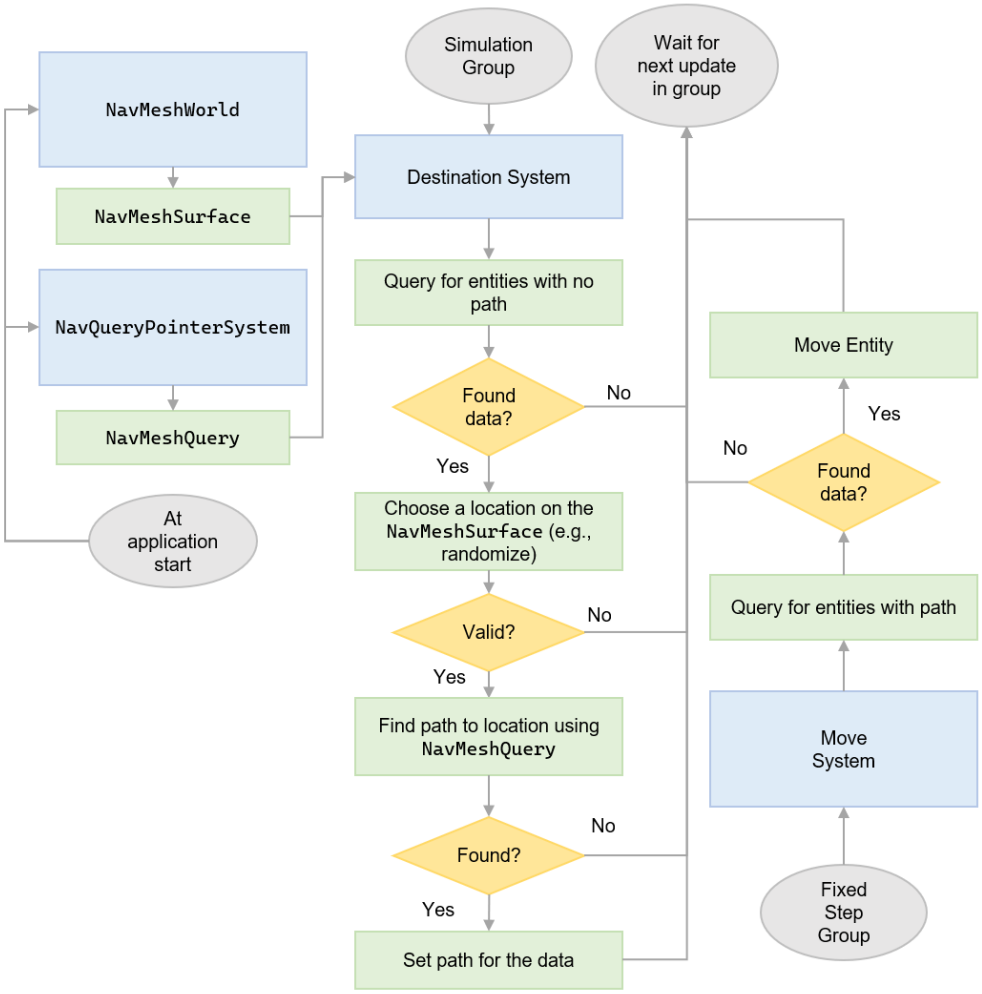

Figure 19 – Pathfinding system activity diagram
Blue = A system, Green = A variable or a function of a system,
Grey = The start or end of an update cycle, Yellow = A Condition node.

5.4.3 Regular Unity pathfinding

As a counterpart to the DOTS pathfinding, an object-oriented pathfinding method was implemented to observe the difference in performance. The method was done using the conventional Unity navigation system. The approach was developed to behave similarly to the DOTS systems, with the

primary difference being that each agent handles their own destination and pathfinding. Figure 20 shows the update `(FixedUpdate)` function of the agents.

```
public class RandomFinderAgent : MonoBehaviour
{
    private NavMeshAgent agent;
    private NavMeshSurface surface;

    private void Start()
    {
        surface = NavMeshSurface.activeSurfaces[0];
        agent = GetComponent<NavMeshAgent>();
    }
    private void FixedUpdate()
    {
        if (agent.remainingDistance < 0.1f)
        {
            agent.SetDestination(RandomDestination());
        }
    }
   …
}
```

Figure 20 – `MonoBehaviour` script used for regular pathfinding.

5.4.4 Animations

DOTS animations were implemented using the sample projects by Unity for the experimental animation package. As the package is experimental, the implementation was not smooth. For example, the animation blending and changing required structural changes, which meant that it could not be run with burst enabled or in parallel. This was a crucial weakness of the experimental package. In addition, some code for the animations was not supported in a standalone build. Specifically, the `BlopAssetStore` class, which was used to get the animation clips for the blending, was not supported.

The animation system first creates an animation graph for the animations. Values of the graph can then be changed to manipulate blending and animation speeds for the units. Units have three animations (i.e., Idle, Walk, Run) that they can change. They also change their animation speed depending on their movement speed. Figure 21 shows the `OnUpdate` method of the `HumanoidAnimPlayerSystem` script with the previous points highlighted.

```
protected override void OnUpdate()
{
 CompleteDependency();

 var ecb = ecbSystem.CreateCommandBuffer();

 Entities
.WithName("CreateGraph")
.WithNone<HumanoidAnimClipStateData>()
.WithoutBurst()
.WithStructuralChanges()
.ForEach((Entity e, ref Rig rig, ref HumanoidAnimPlayerData
animData) =>
{
var state = CreateMixerGraph(e, graphSystem, ref rig, animData); ecb.AddComponent(e, state);
}).Run();

 Entities
.WithoutBurst()
.WithStructuralChanges()
.WithChangeFilter<HumanoidAnimPlayerData>()
.ForEach
((Entity e, ref HumanoidAnimClipStateData stateData, ref HumanoidAnimPlayerData animData) =>
{

    var set = graphSystem.Set;
    if (animData.animState > 1)
    {

        set.SendMessage(stateData.WalkNode, ClipPlayerNode.SimulationPorts.Clip, animData.ClipIdle);
        set.SendMessage(stateData.RunNode,ClipPlayerNode.SimulationPorts.Clip, animData.ClipIdle);
    }
    else
    {

        set.SendMessage(stateData.WalkNode,ClipPlayerNode.SimulationPorts.Clip, animData.ClipWalk);
        set.SendMessage(stateData.RunNode, ClipPlayerNode.SimulationPorts.Clip, animData.ClipRun);
    }
        set.SetData(stateData.MixerNode, MixerNode.KernelPorts.Weight, animData.mixValue);
        set.SetData(stateData.WalkNode, ClipPlayerNode.KernelPorts.Speed, animData.animSpeed);
        set.SetData(stateData.RunNode, ClipPlayerNode.KernelPorts.Speed, animData.animSpeed);
}).Run();
}
```

Figure 21 – `OnUpdate` method of `HumanoidAnimPlayerSystem.`

Animations for the mono animation benchmark were implemented using the conventional Unity tools for animations. The animator had a blend tree blending

between the animations depending on the speed value (Figure 22).

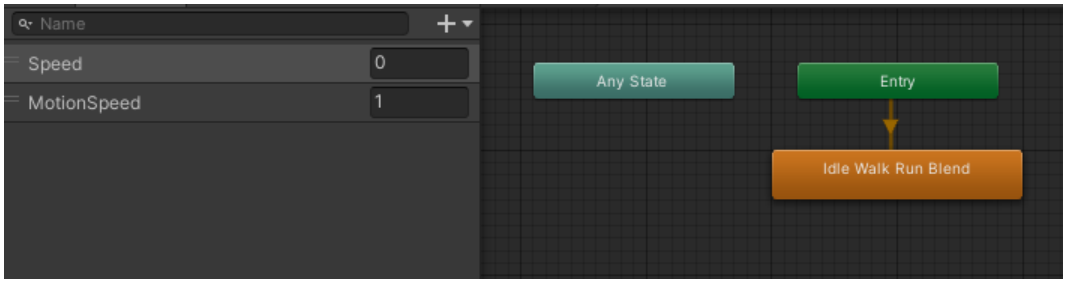

Figure 22 – Mono Animation animator parameters and states.

5.5 VR with DOTS

For VR, the SteamVR plugin for Unity was used. It was chosen as it has vast sample assets and compatibility with the Rift S headset. Same pathfinding as in the pathfinding environment was used except for static destinations instead of randomized.

For the tower defence, a system was created to handle the tower logic. The system used the physics package to find enemy units near the towers. `OverlapSphere` was used to look for physic objects (i.e., enemy units) in the tower range (Figure 23). The system could then apply damage logic onto the found enemy entities.

```
NativeList<DistanceHit> outHits = new NativeList<DistanceHit>(Allocator.Temp);
if (world.OverlapSphere(towerTranslation.Value, tower.range, ref outHits,
    filter))
{
    tower.target = outHits[0];
}
```

Figure 23 – `OverlapSphere` finds units in the tower's range.

5.6 AR with DOTS

Unity's AR Foundation was used for the AR implementation. AR Foundation was developed by Unity to support multiplatform AR development with Unity (Unity 2021a). Same pathfinding as in the pathfinding environment was used for the units. As the units needed a surface to do the pathfinding on, AR foundation

was used to find a plane in the real world using the phone's camera. When the plane was found, a `NavMesh` could be baked on it (Figure 24). Only the plane with the largest area was kept active to make the baking more straightforward. After the baking was complete, the Units did similar pathfinding behaviour as in the pathfinding benchmark environment.

```
private void PlaneManager_planesChanged(ARPlanesChangedEventArgs obj)
{
    foreach (var plane in FindObjectsOfType<ARPlane>(true))
    {
        if (mainPlane == null)
            mainPlane = plane;

        if (mainPlane == plane)
            plane.gameObject.SetActive(true);
        else if (Vector2.SqrMagnitude(mainPlane.extents)
        <= Vector2.SqrMagnitude(plane.extents))
        {
            mainPlane = plane;
            plane.gameObject.SetActive(true);
        }
        else
            plane.gameObject.SetActive(false);
    }
    surface.BuildNavMesh();
}
```

Figure 24 – Script bakes new NavMesh when planes are changed.

6 BENCHMARK RESULTS

This chapter runs through the results of the benchmarks. The next chapter discusses the findings in detail.

6.1 Pathfinding benchmark

Pathfinding benchmarks were run on a desktop and a laptop. Scheduling types were measured to find out the performance difference between parallelized and non-parallelized scheduling types. The highest result of scheduling was compared to the object-oriented results to figure out the performance difference between them. Animated parallel and animated mono were compared to find out the performance difference between animation approaches. The AR benchmark ran on the phones were compared to each other to understand the performance gap between pc and mobile.

6.1.1 Scheduling

In the scheduling benchmark amount of time until fps drops to 10 or below was measured. More time equals better results. Figure 25 (desktop) and Figure 26 (laptop) show the results of the scheduling benchmarks.

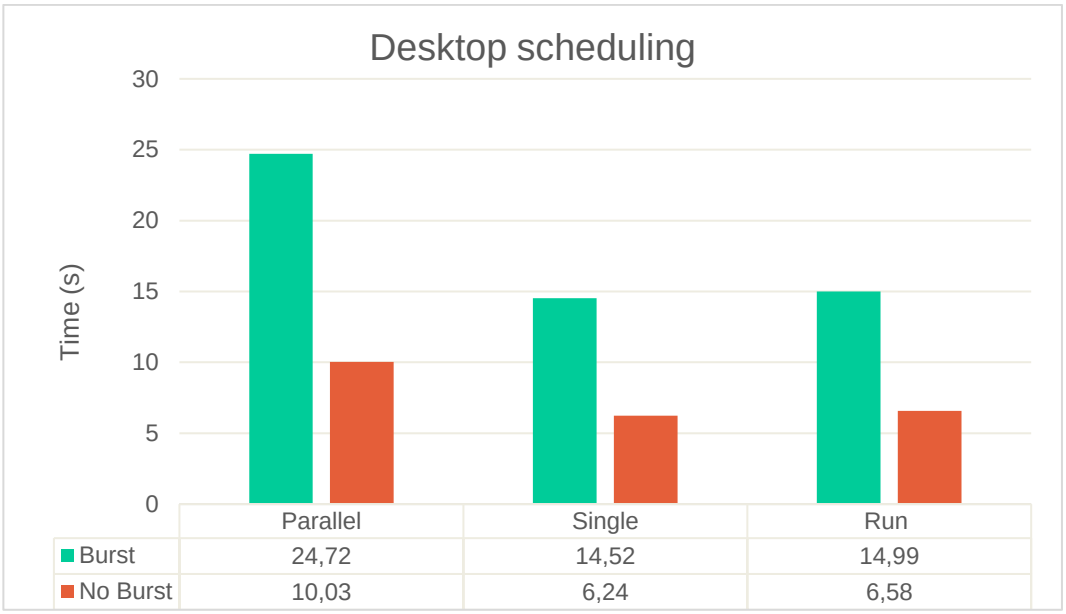

Figure 25 – Desktop scheduling benchmarks.

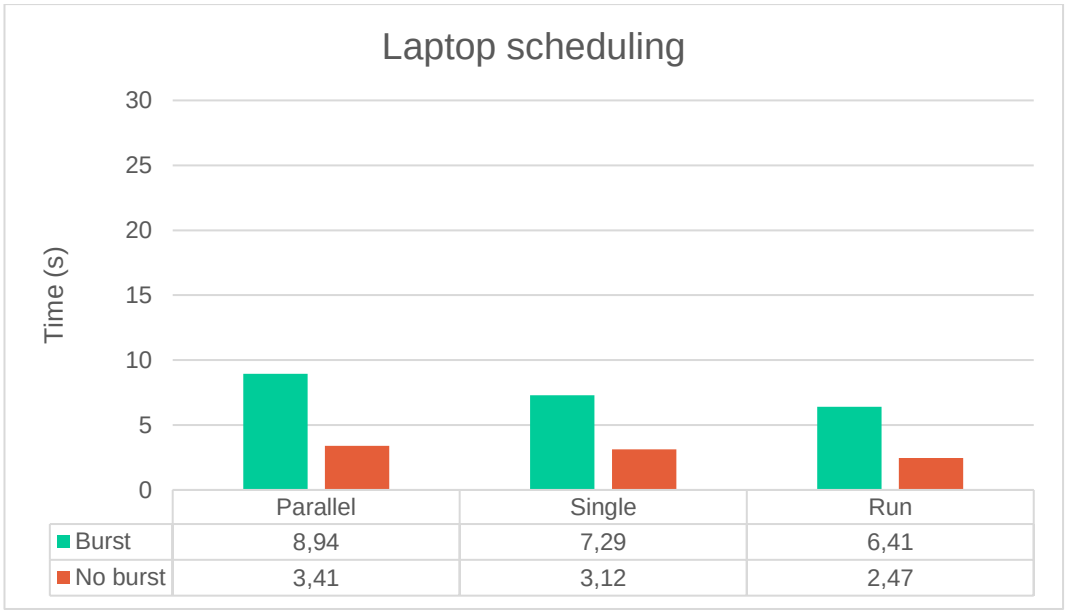

Figure 26 – Laptop scheduling benchmarks.

The desktop was two to three times faster in all benchmarks compared to the laptop. For both devices, the parallel benchmark with burst was the most performant. The burst benchmarks were, on average, 2.5 times faster than their non-burst counterparts. Results for Single and Run benchmarks were close to each other. Run was 3% faster compared to Single on the desktop but 12%

slower on the laptop. The parallel burst benchmark was 68% faster on the desktop compared to single and run. This difference was only 20% on the laptop.

6.1.2 Comparison to object-oriented

The number of units was measured for the comparison as the time measurement was not comparable for these benchmarks. Parallel results are from the pathfinding parallel with burst benchmark, and the mono results represent the object-oriented approach. More units equal better results. Figure 27 shows the results of these benchmarks.

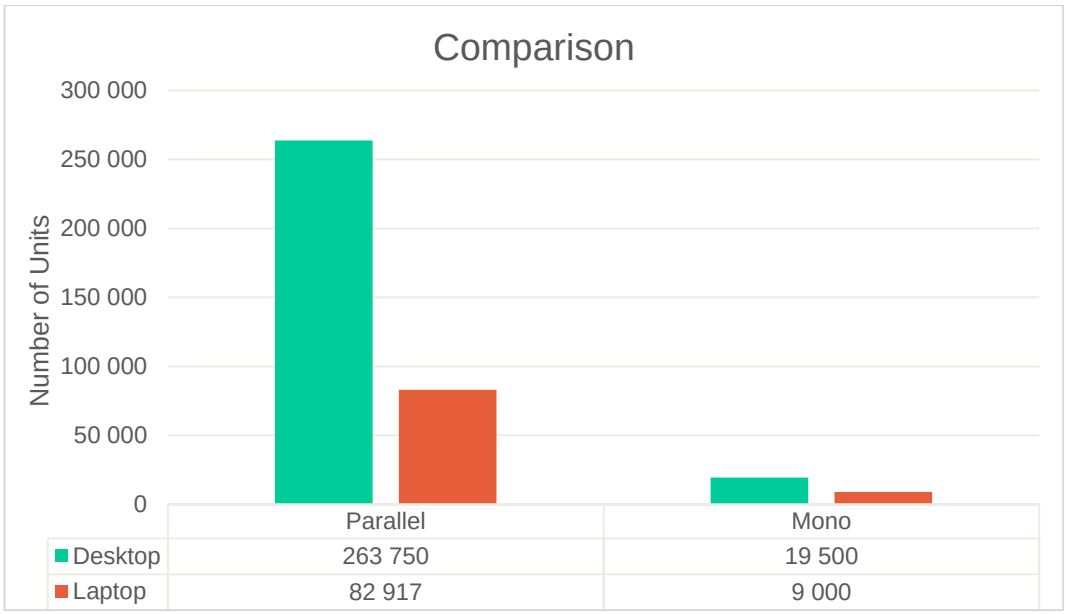

Figure 27 – Comparison between Parallel and mono benchmarking.

Parallel was 13 times faster on the desktop and nine times faster on the laptop.

6.1.3 Animation

Parallel results are from the parallel animated benchmark. Mono results are from the mono animated benchmark. More units equal better results. Figure 28 shows the results of the animation benchmarks.

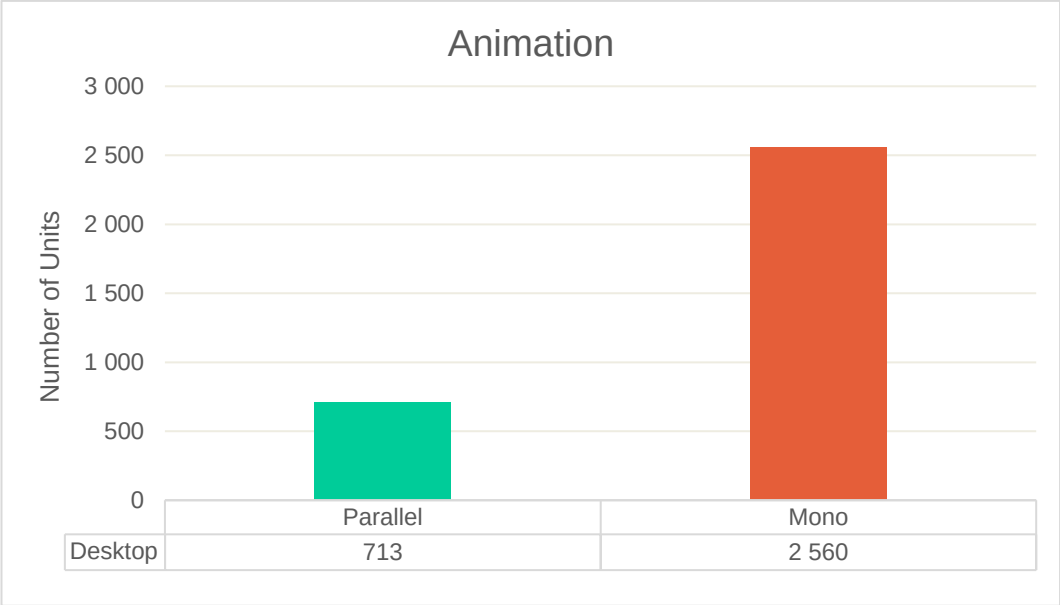

Figure 28 – Animation benchmarks.

The object-oriented approach was 3.5 times faster than the DOTS approach.

6.2 VR

No VR benchmarks were run, but a difference between the performance of the standalone build and the editor was observed. The build performed better than the editor. The build held 40 FPS at 30 000 units while the editor was at 20 FPS with the same number of units.

As the VR was run on the desktop, the previous PC results generally apply to the VR as well.

6.3 AR

In the AR benchmark, the FPS was observed as the number of units increased. The target framerate was 30. More FPS equal better results. Figure 29 shows the results of the AR benchmarks.

Figure 29 – AR benchmarks.

The Nexus started dropping frames at the 1000-unit mark. The Pixel held 30 FPS until 5000 units. At 10 000 units, the pixel performed 6.7 times better, and at 20 000 units 2.7 times better than the Nexus.

7 DISCUSSION & FUTURE WORK

This chapter discusses the results of the research by going through the topics of the research questions. The methods were criticized when it was deemed necessary. Each part provides recommendations and suggestions for future work on their topics.

7.1 Data visualization

There were no problems with visualizing the units and performance data inside the Unity Editor using DOTS. Each unit was rendered and could be observed while running the benchmarks. In addition, the number of units on the screen provided a consistent measurement that could be compared to the other benchmarks. Figure 30 shows an example of the units and the data.

This implementation lacked depth and should be built upon in the future. Outputs such as detailed graphs and data columns should be implemented inside the editor or standalone builds for future work.

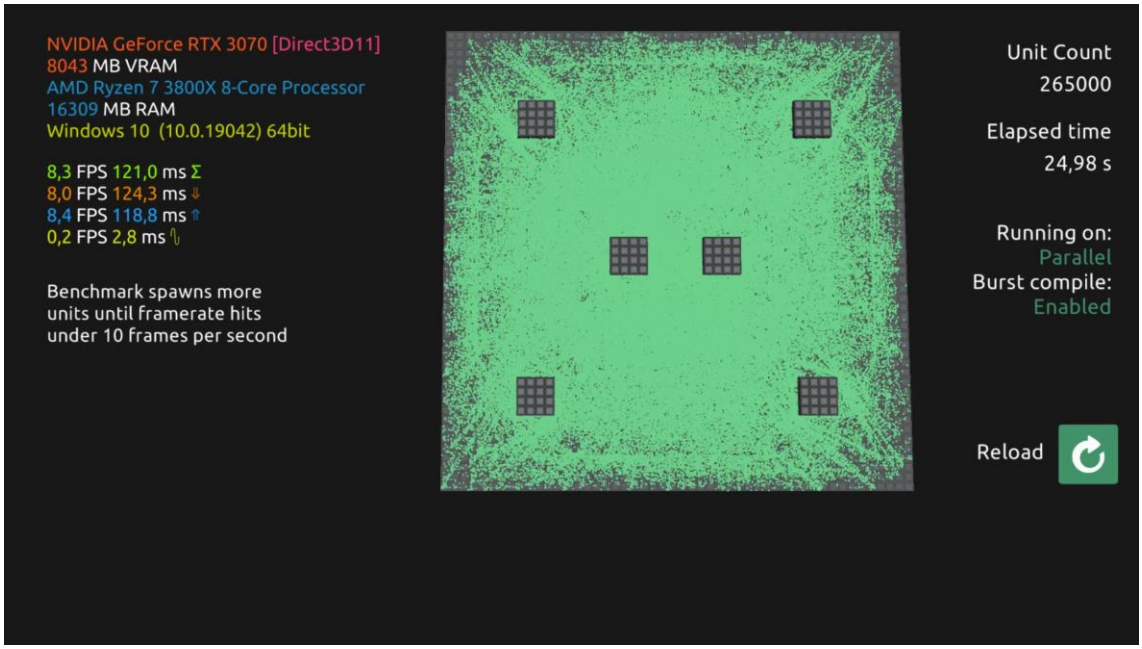

Figure 30 – Example of a pathfinding benchmark run on the desktop where 265 000 units (green shapes) were rendered on the screen.

7.2 Scheduling & burst

As expected, running the benchmarks in parallel with burst was the best performing approach. Compared to single and run scheduling, the performance gains were high (68%) on the 8-core processor but lower (20%) on the 4-core processor. This was expected as the more cores are present, the more the parallel job system can take advantage of them. As the consumers' average processor core count is going up each year (Steam 2021a), these results are promising for the future.

`Single` scheduling was expected to be faster than `run` as it takes advantage of the job system, which was the case in the laptop benchmark but not in the desktop benchmark. `Run` scheduling was 3% faster on the desktop but 12% slower on the laptop. With these results, it is inconclusive which one of them is faster. More devices would need to be tested to get a more conclusive answer.

Research with more processors with more than 8-cores should be concluded, as it would be interesting to see the possible diminishing returns of the added core counts.

7.3 Pathfinding

There were no issues in implementing the pathfinding using DOTS. The DOTS pathfinding performed up to 13 times better when compared to the object-oriented Unity approach. Criticism for the pathfinding methods is the comparability of the implementations. The implementations did not strictly use the same systems, and the DOTS implementations had more time and optimization put into them.

This topic warrants much research in the future. Not just with Unity, but other game engines as well. Research on the performance of different pathfinding algorithms (e.g., A*, Dijkstra, Flow fields) when using data-oriented and object-oriented approaches should be compared inside game engines.

7.4 VR

There were no issues with DOTS compatibility with VR. Its performance was no different to the other performance tests on the same machine. DOTS for VR could be considered for any task applicable in any other PC Unity project.

The methods for testing the VR application were not deep enough as there were no benchmarks run due to weak planning. The object-oriented pathfinding should have been tested with VR as well. This way, the DOTS approach could have been compared to it.

Future work with different VR platforms and plugins should be concluded to confirm the compatibility and performance findings. In addition, it would also be interesting to see if creating a data-oriented plugin to use for VR would be worthwhile.

7.5 AR

As expected, no issues with AR and DOTS compatibility were found. The DOTS pathfinding on the Pixel device performed around as well as the mono pathfinding on the much more powerful desktop machine. This is promising for DOTS as it allows for similar performance on mobile devices as would otherwise only be possible with powerful desktop processors. The slower nexus phone had 37% of the performance of the Pixel, which is still impressive considering the Nexus was released in 2015.

The real-time building of the navigation mesh onto AR planes was an interesting side product of the research. It would be interesting to see if other pathfinding methods can get similar or better results in a dynamically changing area.

7.6 DOTS in production

It was expected that DOTS is not production-ready because Unity themselves had not declared it production-ready yet. Generally speaking, the author agrees with this. The current way of development is not intuitive for someone from the object-oriented world and requires much adjustment. The main factors contributing to this are the usage of data components, limited datatypes, job scheduling and querying entity `ForEach` job. Additionally, the developer will need some knowledge on memory allocation, which is not required in object-oriented Unity development. It was too easy to create memory leaks in DOTS accidentally.

With these points, the author would not recommend relying on DOTS for the most critical features of the project. Some more minor features, such as moving background characters, could be implemented with DOTS. Even then, the time required for learning DOTS needs to be considered when starting production. The author could recommend DOTS for some mobile tasks as battery savings are a prominent topic in that field and would warrant more time taken for the development.

The more time is invested into learning DOTS; the results would undoubtedly also get better. However, as the time for this research was limited, the author's results should be taken from the angle of an object-oriented Unity developer with no previous DOTS experience.

An example of DOTS in action is Vedelem: The Golden Horde by Breda University of Applied Sciences (Steam 2021b). It is a real-time strategy game set in 13th century Europe. The game uses Reese's DOTS navigation (Schultz 2021) for the movement of its units. The game works well with DOTS and should be considered a showcase of what can be created using the current DOTS version.

Unity is still updating DOTS, and the author would advise waiting for the production-ready version to come out before creating important features using

it. Unfortunately, there is no clear roadmap from Unity for when this might be coming out. As it stands, DOTS is worth trying out but not production-ready yet.

In the future, when Unity releases a new version of the DOTS packages, a new test should be concluded to test if the newer iteration has gotten better usability for the developer, which might make it deemed production-ready.

7.6.1 Animation

The usage of the animation is rough compared to the conventional Unity animation tools. The package is in an experimental stage, so this was to be expected. Documentation for the package is close to non-existent, with the sample project being the only resource from Unity to help development. The object-oriented animations run 3.5 times faster than the DOTS animations, even with DOTS having up to 13 times the performance in the non-animated benchmarks. These results might be because of a lacklustre implementation of the animations on the authors part or because the package is not mature enough. As it is, the author does not recommend using the animation package.

8 CONCLUSION

In this thesis, the primary question of discussion was whether DOTS is production-ready. The secondary question was to find out whether DOTS was compatible with VR and AR.

The differences of DOTS to conventional object-oriented Unity development were reviewed and compared through discussion and benchmarks. These benchmarks were realized using pathfinding methods to visualize the data for the DOTS and the object-oriented approaches. The pathfinding was also applied in VR and AR to find out if they were compatible with DOTS. Finally, recommendations on the production readiness of DOTS were given based on the results and findings of these methods. Some points for possible future work on the topics were provided as well.

The discussion concluded that DOTS is not production-ready yet, because the author considered the current DOTS development model too arduous to use. The predominant factor to this was the unfamiliar and complex code syntax that DOTS requires. However, the performance of the DOTS approach compared to the object-oriented approach was deemed promising for the future. Moreover, VR and AR did not have any combability issues with DOTS. The author remarks that DOTS should be researched again when Unity updates and moves it closer to official production readiness.

REFERENCES

Acton, M., 2014. CppCon 2014: Mike Acton "Data-Oriented Design and C++". [online] Available at: <https://www.youtube.com/watch?v=rX0ItVEVjHc> [Accessed 1 September 2021].

Bad Graphix, 2020. How many Boids can Unity handle? (ECS & DOTS). [online] Available at: <https://www.youtube.com/watch?v=mNZq0RhM-98> [Accessed 1 August 2021].

Borufka, R. 2020, "Performance testing suite for Unity DOTS", Master thesis, Charles University, Prague.

Cardelli, L., 1988, March. Types for data-oriented languages. In International Conference on Extending Database Technology (pp. 1-15). Springer, Berlin, Heidelberg.

Codemonkey, 2021. Pathfinding in Unity ECS! (Epic Performance!). [online] Available at: <https://www.youtube.com/watch?v=ubUPVu_DeVk> [Accessed 1 August 2021].

Dijkstra, E.W. 1959, "A note on two problems in connexion with graphs", Numerische mathematik, vol. 1, no. 1, pp. 269-271.

Doucet, L., 2021. Game engines on Steam: The definitive breakdown. [online] Game Developer. Available at: <https://www.gamedeveloper.com/business/game-engines-on-steam-the-definitive-breakdown> [Accessed 1 October 2021].

Fabian, R. 2018, Data-oriented design, R. Fabian.

Hennessy, J. L., & Patterson, D. A. (2017). Computer architecture: a quantitative approach. Cambridge, MA, Morgan Kaufmann.

Hybesis, 2020. Pathfinding algorithms : the four Pillars.. [online] Medium. Available at: <https://medium.com/@urna.hybesis/pathfinding-algorithms-the-four-pillars-1ebad85d4c6b> [Accessed 1 August 2021].

Joshi, R., 2007. Data-oriented architecture: A loosely-coupled real-time soa. whitepaper, Aug.

Krishnaswamy, N. 2009, "Comparison of efficiency in pathfinding algorithms in game development", Honors Senior Thesis, DePaul University, Chigaco.

Llopis, N., 2009. Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP) – Games from Within. [online] Gamesfromwithin.com. Available at: <https://gamesfromwithin.com/data-oriented-design> [Accessed 1 September 2021].

Männistö, T. 2020, "Data-oriented technology in Unity game engine ", Bachelor's thesis, JAMK University of Applied Sciences, Jyväskylä.

Microsoft, 2021a. Structs - C# language specification. [online] Docs.microsoft.com. Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/structs> [Accessed 1 September 2021].

Microsoft, 2021b. Unsafe code, pointers to data, and function pointers. [online] Docs.microsoft.com. Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/unsafe-code> [Accessed 1 September 2021].

Microsoft, 2021c. Object-Oriented Programming (C#). [online] Docs.microsoft.com. Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/oop> [Accessed 1 October 2021].

Mutel, A., 2018. Alexandre Mutel — Behind the burst compiler, converting .NET IL to highly optimized native code. [online] Available at: <https://www.youtube.com/watch?v=LKpyaVrby04> [Accessed 1 October 2021].

Oculus, 2021. VR Performance Optimization Guide | Oculus Developers. [online] Developer.oculus.com. Available at: <https://developer.oculus.com/documentation/native/pc/dg-performance-opt-guide/> [Accessed 1 October 2021].

PassMark Software, 2021. PassMark Android Benchmark Charts. [online] Androidbenchmark.net. Available at: <https://www.androidbenchmark.net/> [Accessed 1 September 2021].

PassMark Software, 2021a. PassMark Software - CPU Benchmarks - CPU Popularity in the Last 90 Days. [online] Cpubenchmark.net. Available at: <https://www.cpubenchmark.net/share30.html> [Accessed 1 September 2021].

Permana, S.H., Bintoro, K.Y., Arifitama, B. & Syahputra, A. 2018, "Comparative analysis of pathfinding algorithms a*, dijkstra, and bfs on maze runner game", IJISTECH (International J.Inf.Syst.Technol., vol.1, no.2, p.1).

Reynolds, C.W. 1987, "Flocks, herds and schools: A distributed behavioral model", Proceedings of the 14th annual conference on Computer graphics and interactive techniques, pp. 25.

Schultz, R., 2021. Reese's DOTS Navigation. [online] Available at: <https://openupm.com/packages/com.reese.nav/> [Accessed 1 August 2021].

Steam, 2021a. Steam Hardware & Software Survey. [online] Store.steampowered.com. Available at: <https://store.steampowered.com/hwsurvey/cpus/> [Accessed 20 August 2021].

Steam, 2021b. Vedelem: The Golden Horde on Steam. [online] Store.steampowered.com. Available at: <https://store.steampowered.com/app/1517150/Vedelem_The_Golden_Horde/> [Accessed 1 August 2021].

Tcha-Tokey, K., Loup-Escande, E., Christmann, O. & Richir, S. 2017, "Effects of interaction level, framerate, field of view, 3D content feedback, previous experience on subjective user experience and objective usability in immersive virtual environment", International Journal of Virtual Reality, vol. 17, no. 3, pp. 27-51.

Unity, 2017. [online] Unite Austin 2017 - Massive Battle in the Spellsouls Universe. Available at: <https://www.youtube.com/watch?v=GEuT5-oCu_I> [Accessed 1 September 2021].

Unity, 2020a. Hybrid Renderer | Hybrid Renderer | 0.11.0-preview.44. [online] Docs.unity3d.com. Available at: <https://docs.unity3d.com/Packages/com.unity.rendering.hybrid@0.11/manual/index.html> [Accessed 1 September 2021].

Unity, 2020b. Using Entities.ForEach | Entities | 0.9.1-preview.15. [online] Docs.unity3d.com. Available at:

<https://docs.unity3d.com/Packages/com.unity.entities@0.9/manual/ecs_entities_foreach.html> [Accessed 1 September 2020].

Unity, 2020c. Entity Component System | Entities | 0.17.0-preview.42. [online] Docs.unity3d.com. Available at: <https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/index.html> [Accessed 1 August 2021].

Unity, 2020d. Burst User Guide | Burst | 1.4.11. [online] Docs.unity3d.com. Available at: <https://docs.unity3d.com/Packages/com.unity.burst@1.4/manual/index.html> [Accessed 1 August 2021].

Unity, 2021a. About AR Foundation | AR Foundation | 4.1.7. [online] Docs.unity3d.com. Available at: <https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@4.1/manual/index.html> [Accessed 1 August 2021].

Unity, 2021b. DOTS packages| Unity. [online] Unity. Available at: <https://unity.com/dots/packages> [Accessed 1 August 2021].

Unity, 2021c. Unity and Havok Physics for DOTS-based projects | Unity. [online] Unity. Available at: <https://unity.com/unity/physics> [Accessed 1 October 2021].

Unity, 2021d. What is DOTS and why is it important? - Unity Learn. [online] Unity Learn. Available at: <https://learn.unity.com/tutorial/what-is-dots-and-why-is-it-important?uv=2021.1> [Accessed 1 October 2021].

Unity, 2021e. NavMeshQuery. [online] Available at: <https://docs.unity.cn/2021.2/Documentation/ScriptReference/Experimental.AI.NavMeshQuery.html> [Accessed 1 August 2021].

Unreal, 2021. Unreal Engine 4 Terminology. [online] Docs.unrealengine.com. Available at: <https://docs.unrealengine.com/4.27/en-US/Basics/UnrealEngineTerminology/> [Accessed 1 September 2021].

Yegulalp, S., 2021. What is LLVM? The power behind Swift, Rust, Clang, and more. [online] InfoWorld. Available at: <https://www.infoworld.com/article/3247799/what-is-llvm-the-power-behind-swift-rust-clang-and-more.html> [Accessed 1 August 2021].

Zeng, W. & Church, RL 2009, "Finding shortest paths on real road networks: the case for A", International Journal of Geographical Information Science, vol. 23, no. 4, pp. 531-543.