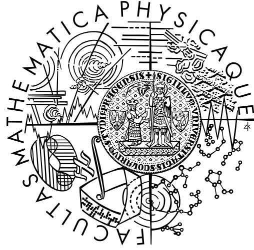

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Šimon Tichý

The Last Clan - RTS game in Unity

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: RNDr. Jan Pacovský
Study programme: Computer Science (B1801)
Study branch: Software and Data Engineering

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I thank my family for standing by my side during times of debugging.

Title: The Last Clan - RTS game in Unity

Author: Šimon Tichý

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Jan Pacovský, Department of Distributed and Dependable Systems

Abstract: This thesis explores the development of a real-time strategy (RTS) game using Unity's Data-Oriented Technology Stack (DOTS) and the C# programming language. RTS games offer captivating real-time battles, requiring players to control multiple units with distinct traits. Traditional object-oriented design often leads to redundant data in memory, but DOTS presents a new data-oriented architectural style that enhances game design.

The goal is to build a game environment where a player can control his clan units, capable of building, gathering, and fighting against computer-driven enemies. The thesis highlights the benefits of ECS through DOTS, such as efficient memory utilization and support for multithreaded code. Through this study, we demonstrate the potential of data-oriented technology, a new approach to designing RTS games, addressing exciting challenges encountered during development.

Keywords: Unity, Entity Component System, DOTS, C#, Unity JobSystem, RTS 2D game

Contents

1 Introduction **3**
- 1.1 Gameplay . . . 3
- 1.2 Thesis structure . . . 4

2 Background **5**
- 2.1 Real-time strategy games . . . 5
- 2.2 Unity engine . . . 5
- 2.3 Unity DOTS . . . 7
  - 2.3.1 Entity Component System . . . 7
  - 2.3.2 Burst . . . 7
  - 2.3.3 Job System . . . 7
- 2.4 Entity Component System (Unity DOTS) . . . 8
  - 2.4.1 Entity . . . 8
  - 2.4.2 Component . . . 9
  - 2.4.3 System . . . 11

3 Analysis **13**
- 3.1 Engine and technology selection . . . 14
- 3.2 Game map . . . 15
- 3.3 Unit navigation . . . 17
- 3.4 Map saving . . . 19
- 3.5 Pathfinding . . . 19
- 3.6 Unit information sharing . . . 22

4 User's documentation **23**
- 4.1 Setting up a game . . . 23
- 4.2 Gameplay . . . 24
- 4.3 Map editor . . . 27

5 Programmer's documentation **29**
- 5.1 Environment preparation . . . 30
- 5.2 The Game Scene lifecycle . . . 30
- 5.3 Map creating . . . 32
- 5.4 User Interface . . . 34
- 5.5 Menu and Map editor . . . 36
- 5.6 ECS Entities and Components . . . 37
  - 5.6.1 Archetypes and instancing Entities . . . 37
- 5.7 Entity Component Systems . . . 39
  - 5.7.1 Pathfinding System . . . 40
  - 5.7.2 Attacking System . . . 44
  - 5.7.3 Building System . . . 44
  - 5.7.4 Gathering System . . . 45
  - 5.7.5 Unit manager . . . 47

6 Conclusion **49**

Bibliography **50**

List of Figures **55**

List of Abbreviations **57**

A Table of controls **58**

B Attached CD **59**

1. Introduction

One of the most exciting computer game genres is real-time strategy (RTS), where players engage in captivating real-time battles. The world of RTS games has seen numerous implementations; however, traditional object-oriented design may result in redundant data occupying memory. In response, Unity has introduced the innovative Data-Oriented Technology Stack (DOTS), which offers a data-oriented architectural style that has the potential to enhance game design. By utilizing efficient data manipulation for entities sharing similar traits, this study embarks on an exploration to develop an RTS game, documenting the challenges and decisions encountered along the way.

Our primary goal is to develop an RTS game within Unity, utilizing the power of the C# programming language and the Unity DOTS framework. Unlike turn-based strategy games, RTS offers real-time decision-making with a dynamically changing number of units, making it an ideal playground to explore the potential of DOTS.

The choice of the RTS genre is not merely for its ability to control multiple entities, each with unique traits, but also because it presents intriguing subproblems like pathfinding, concurrent data management, user input handling, and UI rendering across different scenes. Our decision to embrace DOTS over conventional object-oriented design is rooted in its data-oriented approach, enabling efficient memory utilization and multithreaded code support through Unity's Job System.

Entity Component System (ECS) [20] is a data-driven framework working over unmanaged Entities alternative to the typical Game Object architecture. The three principal parts of ECS are Components representing the data, Entities serving as an identification of Components enabling us to logically group data together, and Systems adding logic/behavior to the Entities by transforming their Components' data.

1.1 Gameplay

The game includes basic settings, Map Editor, and an option to start a new game on a custom map with a computer as an enemy or a teammate.

The gameplay consists of a 2D map with fixed boundaries where a player can control his units, capable of fighting, gathering, and even building. The player needs food and other resources to survive and further develop.

The food is obtainable via fishing on the sea, where resources like minerals and wood are accessible and gathered on the land. Wood and minerals are essential for building construction, where each building offers different traits.

To win the game, the player needs to either destroy the enemy's army or let the enemy starve with a blockade over resources essential for survival.

1.2 Thesis structure

In the Introduction, we introduce the RTS genre and its core mechanics. The Introduction further describes a unity engine with its new data-oriented technology stack and what are the benefits of implementing DOTS' Entity Component System design.

In the Analysis, we further decompose the problems of technology selection, environment/map creation, and information/state sharing amongst units and Systems. Furthermore, we will discuss the pitfalls and architectural choices for pathfinding and its algorithm selection and data representation.

In the User documentation, we will walk a user through the first game launch process, present the basic game mechanics, and give guidance to create a custom map using the included map editor.

In the Programmer documentation, we dig deeper into the environment's basic building blocks like map creation, interface, selected ECS design, and essentially: the Components forming every unit and Systems working over the game data transforming all Entities. Concretely, we will explain the leading Systems responsible for Pathfinding, Attacking, Building, Gathering, and Unit management.

2. Background

In this chapter, we will introduce the RTS genre and the traits of a typical RTS game. Then we will describe the Unity engine, its structure, and the objects it uses. Later in Background, we will discuss DOTS and technologies tied to this stack: Entity Component System, Burst, and the Job System. Ultimately, we will dig deeper into ECS concepts and their internal workings.

2.1 Real-time strategy games

Game franchises like Warcraft, Age of Empires, StarCraft, and Stronghold series belong to the RTS (real-time strategy) game genre. The real-time game means that the game does not run in separate moves like turn-based games, but rather everything happens simultaneously [44]. Thus, it is required for the player to be able to adapt an existing plan or come up with a completely new strategy swiftly as the game moves on.

The typical game environment has almost always definitive boundaries and habitually consists of units, buildings, and resources to gather. The usual goal is to destroy all enemy player units and resources before the enemy does the same to you. The manner to achieve victory in RTS games differs, yet among typical traits lies building Systems, resource gathering, and combat Systems.

The building Systems offer the player to construct different buildings, where the attributes may vary (health, appearance, or cost). For example, barracks might train new soldiers, but warehouse offers storage for gathered resources. These buildings usually represent the player's base of operation. Destroying their preponderance almost always results in a player's eventual loss.

Resource-gathering Systems are crucial in the RTS genre because every action (training a unit, constructing a building, upgrading . . .) costs a particular number of in-game resources. Thus, it is essential to gather more so we can build better to have a greater army and eventually win the game. A player can obtain resources using gatherer units, where the units' ability to gather might differ on specific resources. For example, a shipping unit may fish on the seas but cannot mine in the mines as other gatherer units might.

Combat Systems offer tools for a player to achieve victory, whether via sieging the enemy's base that completely depletes their resources, hunting animals, or simply crushing your opponents in a direct battle. As a rule, the army consists of many unit types with attributes upgradable during the game (such as damage, armor, or a new ability). However, all units come at some cost, whether it is a one-time payment only (building a ship; or training a solider) or they need to be fed and maintained periodically during the game, thus withdrawing your resources constantly.

2.2 Unity engine

Unity is a game engine developed by Unity Technologies. Since 2005, when it was first released, the Unity game engine has supported cross-platform development

backing both 3D and 2D graphic projects. Moreover, with its tools, Unity is nowadays used not only in the game industry but, with its support for augmented and virtual reality, it can be used in the film, engineering, or even automotive industry.

The engine internally runs on native C/C++ but implements .NET and C# wrapper for creating Scripts [47], where the base class from which most Unity scripts derive is MonoBehaviour class.

Scripts are primarily used for player controls or as Components to describe the associated Game Object logic. However, not all Scripts are limited to Game Objects; some Scripts might render textures, implement AI logic, or produce multithreaded code via the Job System. For the Job System and multithreading, we will later introduce the SystemBase class.

As an integrated development environment (IDE) and debugger, Unity supports Visual Studio, Visual Studio Code, and JetBrains Rider.

Assets [1] depict all the items used in Unity projects. Assets can represent visual or audio elements like models, textures, sprites, sound effects, or music. Assets can also describe more abstract items such as color gradients, animation masks, or arbitrary text or numeric data. Furthermore, the Assets are shareable via The Unity Asset Store, where the Unity community offers Assets varying from textures and models to animation and tutorials.

Scenes [46] are also Assets representing the main workspace in Unity and unite other Assets like cameras, user interface, level design, lights, and more. A simple game might consist of two or just one Scene: one for in-game content and the other for a menu. However, having all levels in a single Scene environment can sometimes be challenging. In that case, it is reasonable to split different levels into separate Scenes, each with its unique habitat, characters, and design.

Game Object [25] is the elementary object in Unity containing Components capable of holding data that further determine the object's purpose.

To make an example of a Game Object: an object might be a game camera with Components [8] like Transform, camera settings, and panning script. The Transform Component usually contains position vectors, where the camera settings can contain variables with panning speed, camera zoom scale, etc. And the panning script would be responsible for in-game control of a camera.

The Game Objects always contain the Transform Component - others are optional. Game Objects containing only the Transform Components can serve as a container for other Game Objects in the `Game` Scene hierarchy. We can think of a Game Object as a class in objective programming and Components as its attributes, fields, or even methods in the form of a Script.

The Prefab System allows saving Game Objects, their Components, and values as a Prefab. The ThePrefab [42] is nothing more than an Asset that we can instantiate from to create new Prefab instances in the Scene. A practical use case might be when we need to spawn many units of the same type as trees, NPC (a non-player character), effects, and buildings...

Unity also introduces the Package Manager [39], a tool to manage and import downloaded and Asset Store packages. A package is a container for tools, features,

and Assets. It could be, for instance, specialized editors (for text, animation, and more), project templates, runtime tools, and libraries.

2.3 Unity DOTS

Unity's Data-Oriented Technology Stack (DOTS) [15] is a combination of technologies and packages that introduces a data-oriented architecture design, which offers a highly performant and scalable approach to building games in Unity.

Data-oriented design attaches importance to the separation of data and logic that works with that data. This way, the data can be better managed and sorted in memory, thus minimizing accessing times across the game. The DOTS consists of the Entity Component System, Burst Compiler, and C# Job System.

2.3.1 Entity Component System

Entity Component System (ECS) 2.4 is a data-oriented framework that offers an alternative to Game Objects. The ECS comprises three principal parts: Systems, Components, and Entities. The ECS design splits the data and logic into Components and Systems. The Entities replace Game Objects and serve as identifiers of Components belonging together. Further ECS specifications will be explained later 2.4.

2.3.2 Burst

Burst [4] is a compiler primarily designed to work with Unity's Job System. It translates from IL/.NET bytecode via LLVM compiler to highly optimized native code called high-performance C# (HPC#), often working faster than its C++ Game Object equivalent.

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies, where the name itself is not an acronym but the full name of the project. [33]

The benefits of using Unity's Job System are improved application's overall performance. Burst default compiles your code using just-in-time (JIT) compilation as the game runs but also uses the ahead-of-time (AOT) compilation of supported code before the game even starts.

Burst uses the Just In Time (JIT) method to compile the code parts only when the code is required/called. During JIT compilation, bytecode-like code compiles to more optimal machine code at program runtime. The optimizations are possible due to the constant analysis of currently executed code parts and calculating if their optimization would increase the overall performance. The Ahead-of-time (AOT) compilation, contrary to JIT, does the optimization before the program runs rather than at program runtime and compiles to the native code.

2.3.3 Job System

C# Job System exposes Unity's internal C++ Job System, where the CPU threads are managed by the Unity Job System [29] and offers an abstraction

of Jobs. A Job is a small method or unit of work designed for one task and can have an input and output.

One of the main advantages of using Jobs is their ability to be chained so one Job will only run after the dependent Job completes. While writing multi-threaded code, possible race conditions might occur. To monitor and prevent race conditions that might occur while writing multi-threaded code, the Unity Job System has its own Safety system. With this well-designed system, a developer can take the effectiveness of safely parallelized code with minimal performance overhead. The only drawback would be that due to the Safety system, where a Job can only access blittable data types.

A blittable data type, unlike a non-blittable, does not require conversion between managed and native code [16] since it does not need to be processed by the Interop Marshaler due to its common representation in both managed and unmanaged memory.

The Interop Marshaller [34] handles the representation for ambiguous types and gives default representation or alternative representations where multiple representations exist. The Interop Marshaler does this by having multiple instances of the data appearing as a single one.

This process of marshaling is usually automatically invoked by the CLR (Common Language Runtime) [7] and stands convenient not only when working over the same data from multi-threaded code.

The int, byte, single, or double are all blittable types [3]. The blittable type can also be a one-dimensional array of blittable primitive types, like an array of bytes, or a value type containing only blittable types. The non-blittable types are, for example, string, object types, but also bool class.

2.4 Entity Component System (Unity DOTS)

Entity Component System (ECS) [20] is a data-oriented framework available via the Entities package that is obtainable through the package manager. ECS introduces a new data-driven architecture using unmanaged Entities alternative instead of the Game Objects 2.2.

Entity Component design can gain massive performance by optimal memory management over Entities' Components. On the contrary, the Game Objects lay randomly in memory, where the ECS optimizes the Chunks and the Chunk access time. ECS also enables unprecedented control and determinism over its Entities on multiple threads simultaneously.

The three principal parts of ECS are Entities (identity), Components (data), and Systems (logic or behavior).

2.4.1 Entity

The easiest way to think of a simple Entity [19] is a nameless lightweight Game Object with only the *`Translation`* Component associated. An Entity is principally just an ID holding no data and no behavior but instead groups pieces of data (Components) that belong together. The Entity IDs are the only stable way to reference another Component or Entity. Systems then provide the behavior, and Components hold the data.

2.4.2 Component

The ECS Components [17] represent all the different data fields associated with an Entity. The ECS Component is not the same as the Unity Component [58], which is the base class for everything attached to a Game Object. The Unity Component can additionally contain logic, unlike the ECS Components, where the logic is implemented separately in Systems.

The ECS Components either need Garbage collection and are called managed or do not and are called unmanaged. The managed Component can contain any type, unlike the unmanaged Components that contain only blittable and other unmanaged types. However, the benefit of using unmanaged Components over managed ones is that they are accessible in Jobs and Burst compiled code [11].

Component Archetypes and memory layout

A unique set of Components associated with one Entity is called an Entity Archetype [17]. Two Entities are of the same Archetype if they have the same set of Components.

The unmanaged Components are maintained in Chunks. A Chunk is an allocated continuous block in memory dedicated to a single Archetype. A Chunk can only store Entities of the same Archetype. When a Chunk becomes full, ECS automatically utilizes another new block/Chunk, again for a single Archetype. Thus, adding and removing Components of an Entity change the Entity's Archetype, resulting in its relocation in memory to a different Chunk. The Entities in Chunks are not sorted in a particular order because the space in a Chunk is assigned when the first Entity of the corresponding Archetype moves to the Chunk.

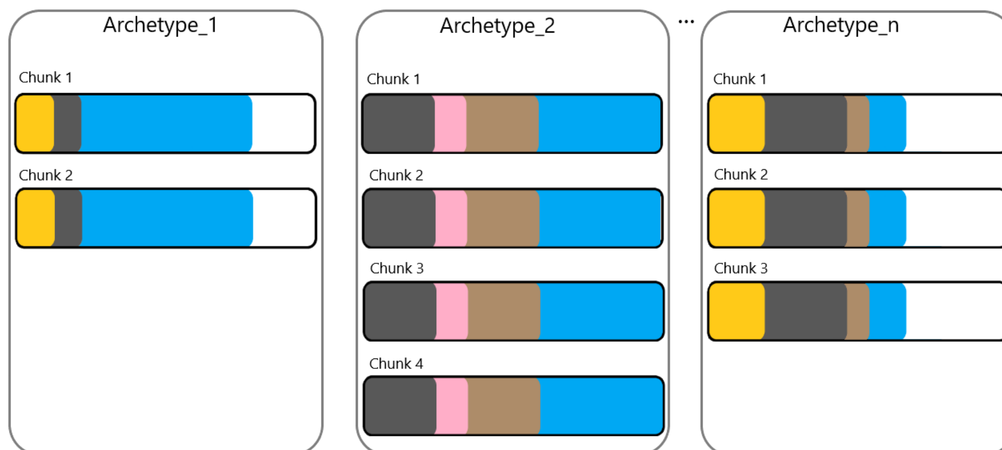

Figure 2.1: Each Archetype has zero or more Chunks, and each Chunk hosts one or more entities of that Archetype. Chunks hosting the same Archetype will thus have the same data layout.

The managed Components are not stored directly in the Chunks. Instead, a World consists of one long array referencing all the class instances. Other managed type Components can store an index to this array. However, accessing a managed Component of an Entity causes some extra overhead due to the necessity

of an extra lookup; this extra work makes them less optimal than the unmanaged Components.

Component manageability and data accesss

To read or write Component values or change or add Components of an individual or multiple Entities, ECS uses an Entity Manager to manage all Entities in a World. Entity Manager also keeps track of all the different Archetypes and organizes the data associated with an Entity for optimal performance. It is also possible to instantiate new Entities from a particular Entity Archetype.

An EntityQuery [23] is a great tool to specify a target Entity or Entities, which is an ECS object we can use to describe an Archetype of an Entity or find a specific Entity with Components matching the input Component requirements (`WithAll`, `WithAny`, `WithNone`). The EntityQuery efficiently retrieves the matching set of Chunks.

Structural changes are not executable in parallel from a Job and must run on the main thread instead. A structural change is, for example: creating and destroying a Chunk or adding and removing Entities from a Chunk.

To make structural changes from a Job/parallel code, the ECS uses an EntityCommandBuffer, which at predetermined moments called sync points [48] runs/performs/*playbacks* [41] all structural changes. A synchronization point (sync point) is a point in the program execution that will wait to complete all already scheduled Jobs. Sync points are thus great to minimize since it limits the benefits of using the parallelized code.

EntityCommandBuffer offers methods for enqueuing all thread-safe actions/ structural changes, processes all the buffered changes, and replays them later on the main thread (also called *Playback*) when a given sync point gets reached, or a Job completes.

To prevent non-deterministic behavior while splitting the recording of commands in EntityCommandBuffer across multiple threads, the ECS uses int sort keys to record the order of all changes. The sort key is passed as the first argument to each ECB method and used to sort the commands before Unity performs the commands [41].

Component Interfaces

Every ECS Component has to implement one of the Component Interfaces: IComponentData, ISharedComponentData, ISystemStateComponentData, ISystemStateSharedComponentData, or IBufferElementData. These interfaces have no methods or properties but mark the struct or class as a type of ECS. The interface also affects the type of a Component, its memory representation, and its manageability by the Garbage collector [17].

The most basic Component type implements the IComponentData interface and can be either managed or unmanaged. The managed IComponentData Components are classes containing fields of any type. The unmanaged IComponentData Components are structs containing only unmanaged or blittable field types. If an unmanaged struct without data fields implements IComponentData, it is called a Tag Component [14] and behaves like a regular unmanaged Component type.

To represent array-like Component structures, ECS offers the DynamicBuffer [10], a resizable array. Its elements are unmanaged struct Components implementing the IBufferElementData interface with the same field constraints as the IComponentData interface. IBufferElementData struct defines the elements of a DynamicBuffer type and the DynamicBuffer Component type itself.

To have a few instances of one Component shared amongst multiple Entities, the ECS introduces Shared Components [12], a struct implementing the ISharedComponent interface. These Components are shareable between Chunks, and their values are without duplicities. All SharedComponent values are stored as managed objects in an array outside the Chunks. The Chunk stores only one index for each shared Component in its Archetype. Since the values get shared in the array, every change to the array counts as a Structural change.

Similar to Shared Components are Chunk Components [9], but the Component data are shared only within the Chunk, unlike Shared Components, which can share across multiple Chunks.

System State Components [13] implement ISystemStateComponentData and are like regular Components. However, an entity with one or more System State Components is not destroyed but will get specially marked and remove all its non-System State Components.

2.4.3 System

Systems [49] add logic/behavior of an Entity to the ECS workflow by transforming the Components' data. An example of such might be a movement System that calculates the position of an Entity for the next frame.

Unity ECS automatically discovers all in-project System classes implementing the SystemBase and instantiates them at runtime.

Every System script needs to implement the SystemBase class. The SystemBase requires its children to implement OnUpdate() method, which runs every frame as there are Components or Entities that this System changes. The other callback functions are optional; for example, the OnCreate() method can initialize the System because it runs when System gets created, if implemented [50].

The System execution order [51] is determined by the System Group to which the System belongs. The default World contains a hierarchy of ComponentSystemGroup instances. The ECS initially contains three root-level System Groups: InitializationSystemGroup, SimulationSystemGroup, and PresentationSystemGroup. We can also create custom System Groups and decide the order execution amongst System Groups to run before or after other Groups or Systems. The System Groups can also be nested in each other.

System example

An example of an ECS System might be a game containing two tree cutters commissioned to search and cut nearby trees (Figure 2.2). The tree cutters would be Entities, and so the trees. We would split the design into two Systems, one responsible for navigating to the nearest tree and the other for cutting the tree if in range.

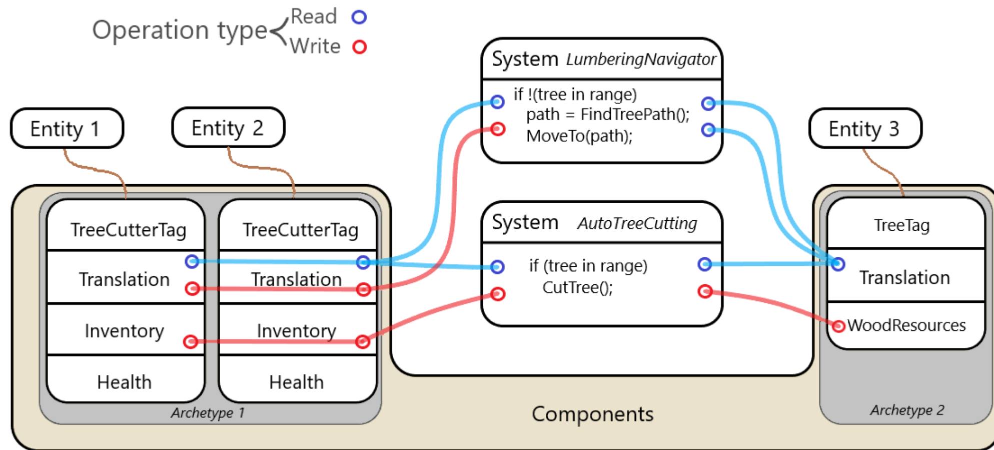

Figure 2.2: System *LumberingNavigator* fetches the positions of trees and navigates the tree cutters to the closest tree by transforming their *Translation*. System *AutoTreeCutting* then cuts the tree redistributing the wood resources from the tree to the tree cutters.

System *AutoTreeCutting* would iterate over all Entities with *TreeCutterTag* and fetch their *Translation* value for comparison with the *Translation* of *TreeTag* Entities to see if any trees are in range. If a tree is in range, its *WoodResources* are changed and added to the Inventory of each's Entity with *TreeCutterTag* gathering nearby. However, if the tree is not within reach, the *LumberingNavigator* System seeks the nearest tree, calculates the path, and moves the cutter to the tree.

Entities 1 and 2 have the same tree cutter set of Components and are thus the same Archetypes. In contrast, Entity 3 represents a tree with a unique set of Components forming a unique Archetype.

Entity 1 and Entity 2 might share the same Chunk since the Chunk hosts multiple data of one Archetype, whereas Entity 3 would be alone in its own Chunk dedicated to Entities of the tree Archetype.

3. Analysis

We have chosen an RTS genre besides others because it offers control over many units capable of interacting with each other in real-time. The topic of managing multiple Entities in a game is interesting because, in the RTS genre, many of them share similar properties. Thus, a well-thought design could save a programmer from overused memory containing a lot of redundant data, which can lead to not optimal access times.

Unity's DOTS is a Data-Oriented Technology Stack introducing a new data-oriented architecture design that further separates the data and the logic. It unifies technologies and packages and answers the problem of data redundancy and manageability over multiple similar data fields and behavior.

Entity Component System (ECS) is a framework inside DOTS and the focus of this thesis. ECS uses unmanaged Entities instead of Game Objects and is available via the Entities package in the Unity engine.

The whole game development consists of many steps, and each step's implementation might vary between companies or developers.

Our game development challenges fall into these logical categories: the User Interface with menu, game Entities and their logic, and the game environment itself. The 2D graphics utilized in the game are licensed under Creative Commons.

The UI is usually implemented either by the new becoming standard UI Toolkit, the Unity UI (also known as uGUI), or the Immediate Mode GUI, but that is rarely used in practical solutions and is most often used just for debugging purposes.

The Unity UI, combined with the TextMeshPro package, has been chosen because it offers almost the same functions as the UI Toolkit [56]. The main drawback of the UI Toolkit is that some parts and features need to contain stable implementation.

Units, buildings, resources and their logic are in traditional Game Object-oriented design, represented as Game Objects with Components containing its attributes, where an attached Script gives behavior to that Game Object. For example, a tree Game Object would have a Component position and a Component holding number of resources associated with this tree. However, the tree logic, like leaves falling, would be included in the attached Script component implementing the MonoBehaviour class.

The ECS uses an Entity instead of the Game Object; the tree would be an Entity with a Component ***Translation*** containing position and a Component *NumberOfLogs*. The leaf-falling logic would be separated in a System script implementing the SystemBase class.

The controllable units in the game have Components like health points and attack power/range. Different types of Units can do various actions like gather or swim. The ECS technology then ensures better manageability of the units and effectively iterates over example Archetype-specific data fields (like ***NumberOfLogs*** obtainable from all trees rapidly).

The game environment is designed to represent a 2D map with two main surfaces `Water` (containing `Fish`) and `Land`. Further, The `Land` hosts `Mineral-Deposits`, `Grass`, and `Rough terrain` capable of slowing units running over it. `MineralDeposit` and `Grass` serve as possible spawn points for minerals and trees.

The requirements on the environment representation are to quickly refer to all map `Land` and `Water` components without the need for additional actions like raycasting to determine position. It is also essential to correctly represent all obstacles and also be able to determine their exact position and size, but most importantly, to integrate well with pathfinding algorithms essential for the RTS genre.

Features such as Fog of War, deeper economics mechanics, or multiple unit formations are good-looking, but they are just more abstractions. The main goal was to understand and develop an RTS game using the ECS technology's data-oriented design, a relatively new approach to game development.

3.1 Engine and technology selection

Most of today's games get developed under some game engine. A game engine is a software framework that (like other frameworks) can significantly shorten the time of implementing basic logic and usually offers easy-to-manage control over multithreaded code or graphics like rendering settings.

The biggest video game companies usually use their proprietary engine adjusted to their needs with automated patterns for easier development or handy tools and packages. The most recognizable might be Bethesda with the Creation engine, Valve with their Source engine, Rockstar with RAGE, Ubisoft with Anvil (formerly known as AnvilNext), or DICE with the Frostbite engine. In the RTS genre, proprietary engines dominate the market. Noticeable engines might be Genie Engine used for Age of Empires or the Warcraft engine developed under Blizzard Entertainment used for older Warcraft and StarCraft games.

Some features that an engine (Unity including) often offers are a vector system, basic implementation of game physics, networking, and multithreaded programming interface.

The game engine "market" offers dozens of game engines, yet some are under paid license. But to sum up all essential traits like license cost, documentation, and support: there are two most recommended options Unity or Unreal engine.

The Unreal engine, as well as the Unity engine, has support for visual scripting and runs internally on C++ (Unreal engine, for instance, offers a node system where a programmer can build up the logic by creating a dependency graph with logical nodes...).

We have chosen The Unity engine because: Unity has the DOTS with ECS, Unity uses C# for the scripting language, as opposed to the Unreal engine, which uses C++; and in my opinion, Unity has more readable documentation and a more active community. The community of Unity's forum actively shares experience, sounds, graphics, and Assets containing simple logic and extension packages.

An event-based system using objects is an alternative to the classical DOTS System using ECS. The main drawback is that a larger count of units (= Game

Objects because the event-based system is Game Object-oriented) would imply more frequent event calls being a bottleneck just by technology selection which we want to prevent.

The ECS, a relatively new and remarkable technology, was selected because it offers scalability over many units (especially these sharing similar data instances like health points or team color of a unit) with improved access times over the classical Game Object-oriented design.

Changing and adding Components to an Entity does not affect the other Components' data (which stays in place) because Components are mutually independent, each representing a different attribute. The advantage of Components is their reusability over many similar Entities. An example might be a ***`Heath`*** or HealthPoints (HP) Component (usually two numbers - current HP and max HP) that we can reuse on a building or a soldier Entity because this Component initially represents the same. Thus, we can represent similar traits/attributes over multiple Entities without creating a new type-specific structure holding HP.

The two (minor) reasons to select ECS might offer optional future extensibility. The first is ECS's optimized multiplayer interface and server authoritative Netcode library [37], yet unused in this thesis. The second is ECS's HDRP (high-definition renderer pipeline) [27], which optimizes graphically complex objects due to the well-designed fetching to memory with less memory usage.

Other ECS alternatives to Unity's DOTS ECS (often published on GitHub) usually market themselves as having better performance, readability, or design than others. The most famous might be Entitas [18], an open-source ECS framework having code-independent logic, claiming better performance (than the DOTS ECS has) without sacrificing the code readability.

The drawback is that Entitas documentation is rather brief, the updates are released slowly, and it only provides support via a Discord server.

Unity's DOTS with its ECS is a more suitable tool for this task with its newest stable updates, well documentation, and C# scripting support.

The Unity version selected for this thesis is 2020.3 because it was (at the start of the project) one of the preferable stable versions offering support for ECS 0.51 [19]. An alternative to ECS 0.51 is to use ECS 1.0 [22], a newer yet not-so-stable version.

The ECS 1.0 introduces a new Baking [2] workflow enforcing pure ECS design, replacing the ECS 0.51 workflow of using a converted Game Object Prefab Asset as an Entity Prefab. Unfortunately, it is complicated to migrate the whole project from ECS 0.51 to 1.0 since ECS introduces breaking changes in HDRP, scene management, and other essential parts...

At the time of writing, the 0.51 version is still preferable amongst the community because the 1.0 version has tendencies to crash often, and the DOTS physics package has occasional difficulties working correctly.

3.2 Game map

The map consists of two main surfaces: *land* and *water*. The *land* contains either a *mineral deposit* (spawn point for minerals), *grass* (spawn point for trees), or *rough terrain*.

The game plays on a 2D map with fixed boundaries. The world map with boundaries is a common practice in the RTS genre (as well as 2D design). However, the main reason why we have chosen 2D over 3D is that most problems we have dealt with in 2D contain the same abstract idea that can be transformed likewise to 3D, without some unnecessary difficulties of 3D development (for example, harder pathfinding with different ground levels and overall, more difficult rendering practices which is not the goal of this thesis).

There are more approaches to creating a 2D map. The usual practice is to use a Tilemap [54], 2D objects on a 2D plane, or 3D objects using a top-down look (but that would again include the 3D graphics that might require special attention).

Grids are Game Objects [26] uniting Tilemaps, where each Tilemap holds information about all containing tiles and their location. The Grid affects its Tilemaps' tiles layout and can be isometric, rectangular, or hexagonal. The rendering, shadows, lights, and graphics settings (of a Tilemap) are manageable via the Tilemap renderer [55], which offers control over all tiles from one point.

The Game Object design would require an external array holding the tiles' position data, which was why we picked Tilemap with Rectangular Grid, where the rectangular design offers an intuitive tile position system (each tile is addressable simply by a two-dimensional vector), a handy in RTS for ***Building*** Systems, pathfinding, and the Map editor.

The second reason for using Tilemap is that in Game Object design, each object has a Rigidbody [43] Component, which controls the object's position and collider but presents the Rigidbody Component responsible for Game Object positioning and collision detection, which could be a bottleneck if calculated for too many units.

The other problem might be pattern making (like continuous ***Grass*** or ***Water*** tiles), which is nicely solved in Tilemap but would require a script for placing logic if used with Game Objects. Tiles (rule tiles) placed next to each other can create patterns depending on other tiles' positions, leading to a prettier game and a Map manager.

Every placed tile is stored in a Tile palette [53]. The Tile palette is like a classical palette created by the user containing placeable tiles. There are more types of tiles: basic tiles (just a sprite or image), animated tiles, and rule tiles (the ones used for pattern making), all aligned to a Grid.

The 2D graphics of (animated/rule) tiles used were under CC (Creative Commons) distributable and adaptable licenses from Opengameart.org [38], where the file with all resources is in the document attached to the project.

Upon reconsideration, Tilemaps were deemed unsuitable, and we would not opt for them in future implementations due to their poor scalability. Initially, we intended to accommodate user-defined map sizes. However, our testing revealed that utilizing an initial map size 600+ by 500 tiles resulted in undesirable performance issues such as low fps and tearing. Consequently, we limited the map size to 200x200 to ensure an enjoyable and playable game experience.

3.3 Unit navigation

Navigating through the map for computer-driven units might cause some difficulties. Since every unit is limited on the surface, it got spawned (ship on the ***Water*** area and archers and warriors on the ***Land***); to navigate or populate unexplored territories, we have to introduce some "conquering" tactics. A few tactics to consider: send units randomly, directly or split the map into components, analyze, and only then send the units.

Sending units randomly to "bump" into an unexplored map part could be highly not optimal since the unit could spend vast time trying to find a path to the part of a map where units cannot enter, even tho it is the same surface as the figure 3.1 shows.

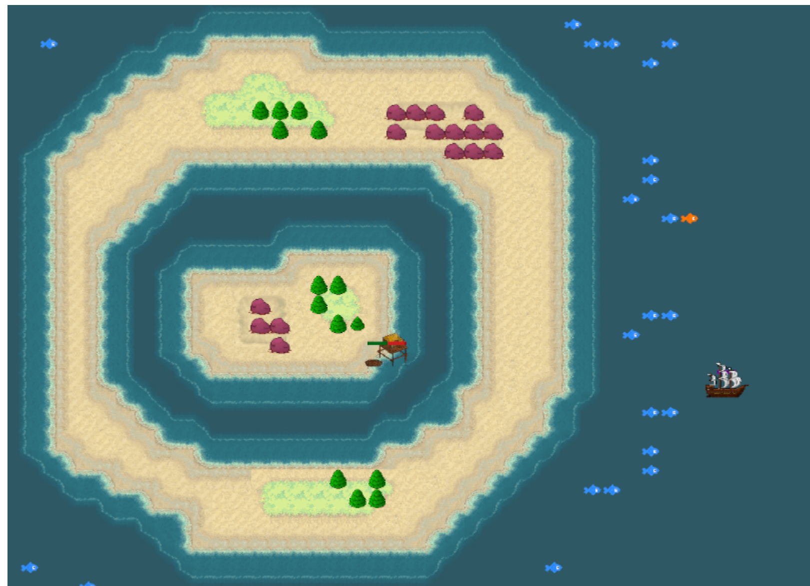

Figure 3.1: The Ship would spend considerable time finding a nonexistent path to the inner ***Water*** circle even though the Ship and destination are the same tile type.

The other method would be to mark the location of enemy units or part of the maps we want to populate is to walk into a straight line, and if the line poses some obstacle, build a Warehouse on it, spawn units that will build Fishery, and so on to overcome all obstacles. Figure 3.2 shows why this implementation could be more optimal.

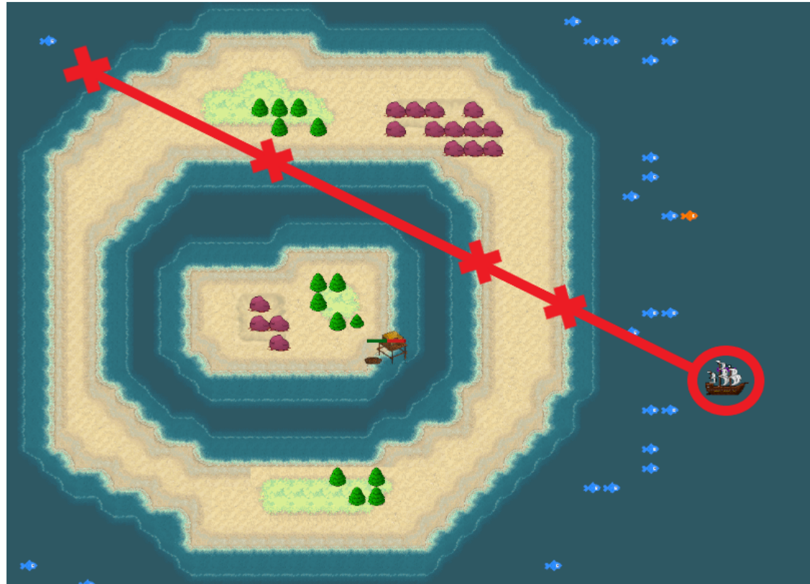

Figure 3.2: Exploring in a straight line could result in unnecessary buildings constructed to overbear components that are unnecessary to pass to get to the destination. Crosses signalize all the buildings built when a player would want to explore using this algorithm.

We have chosen to split the map into so-called components 3.3. If two neighbor tiles are of the same type, they belong to the same component. These components are simply continuous parts of the map with the same tiles. Every component can be referred to by a number which helps us to create an adjacent component graph to register all components sharing at least one neighbor tile. This solution solves both of the mentioned issues of previous implementation choices.

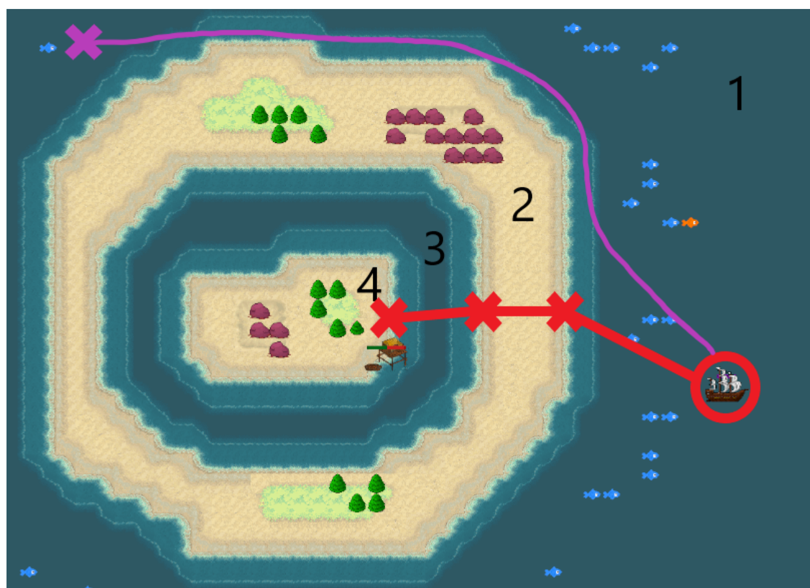

Figure 3.3: In our solution, the map falls into components. With components, a computer-driven clan can determine which components were not conquered yet (marked with red) or if the path is in the same component and exists (marked with purple).

Splitting the map into the components will further help us in the pathfinding unstucking 5.12 or the initial player spawning in the `GameStateManager.cs`.

3.4 Map saving

The Map (Tilemap) consists of many types of surfaces: *land*, *water*, *grass*, *rough terrain*, and *mineral deposit*. The Map terrain needs to preserve this data plus the players' spawn points and load them into a new game or the Map editor. The tamper-proof file format is unnecessary since the game plays without the Fog of War, and all maps are editable.

The preferable saving methods are to use C# binary serialization (with the . NET's Binary Formatter Class) or JSON serialization (via the JsonUtility [31]). Using serialization requires the data to be represented by a serializable object. The object is (then) serialized into a binary file in the case of a Binary Formatter or JSON file in the case of a JsonUtility.

The Binary Formatter and JsonUtility might be the right tools for saving the whole game state: unit position, unit plan, each player's resources, and not yet gathered resources. The drawback is both sterilization methods use way too much data when this problem is nothing more than a 2D array (having the map size) able to store terrain data of each tile according to its position. With the 2D array representation, it is easier to save data into a text file, each char representing one tile (terrain type).

3.5 Pathfinding

Pathfinding is one of the fundamental game mechanics in the RTS, whether it is combat following, walking to gather, or just wandering; it is essential to do it optimally.

Potential problems with data redundancy might arise when multiple selected units get an order to move to one location. The target destination is the same for all Units, yet their individual paths to reach this goal are not always that different.

Land pathfinding: The usual choice of algorithm for game pathfinding is A* (A-Star), but we chose something other than this option (at least for ***`Land`*** navigation) because it has the two following drawbacks.

The main problem is that as the game moves on, the map change (construction and destroying of buildings, cutting trees). The continual change keeps marking some calculated paths as obsolete, thus raising the necessity of recalculating them in real-time. Frequently recalculating might be costly in a solution where each unit keeps track of its path nodes.

The other problem is that an algorithm selected must consider even forces between units (so they would not get pushed away from the calculated Path) as Figure 3.4 describes. A solution using the A* algorithm would demand recreating the Path again if the units got away (e.g., pushed by other units) from the precalculated path nodes.

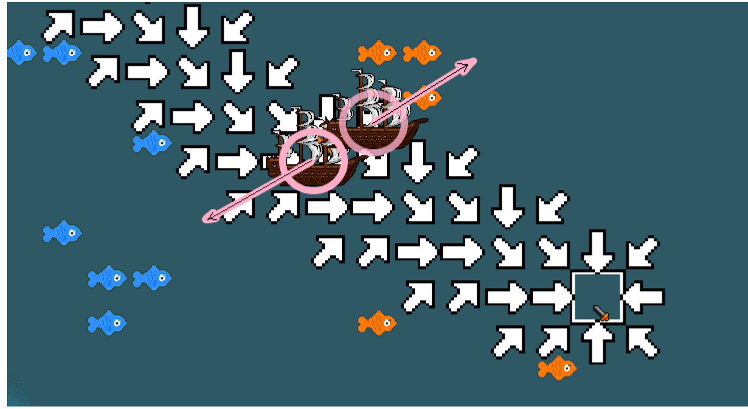

Figure 3.4: Suppose a Ship (lower one) gets scheduled with a path and encounters another friendly Ship(upper one). In that case, forces between units (pink arrows) could throw the first ship off the precalculated path, and the path would need to be unnecessarily recalculated. That is the reason why forces between Ship units are so small.

We can count more tiles around, but sometimes that would be unnecessary, and it still would not be guaranteed that these cases would not happen (potentially leading to performance inconsistency).

That is why we have chosen *Flow-field* (*Vector-field*) pathfinding. *Flow-field* pathfinding resolves the shortest path task from all tiles on the map to the target. All units with the same target have reference to this one ***FlowField*** stored in their ***Target*** Component, and any change in the map would affect only one ***FlowField***.

The *Flow-field* pathfinding also allows forces between units since a unit pushed to another tile still has information about the shortest path from that tile, and there is no need to find to calculate a new one.

Figure 3.5: *Flow-field* pathfinding

The problem with *Flow-field* is that sometimes we calculate paths from tiles that will never host units; these calculations would be unnecessary, increasing our computation time. This issue is amplified on the ***Water*** since *water* surface counts as the majority in my game and usually occurs in larger continuous blocks than ***Land***.

Water pathfinding: We have decided to make A* pathfinding for ships because they have fewer collisions/ interactions than ground units due to the small spacing forces between ships. ***Water*** resources (fishing spots) are not qualified as obstacles, so the updates are similarly less frequent. The ships also cost more resources than ***Land*** units and thus would appear in the game in fewer numbers than ground units, which usually form larger groups.

Auto-gathering pathfinding: The other issue with data redundancy while pathfinding is auto-gathering. Units flagged with ***AutoGatherTag*** automatically migrate from resource and deposit building; on ***Land***, it is wood and minerals deposited into the Warehouse; on ***Water***, it is food from fishing spots deposited to the Fishery.

Figure 3.6: ***Land*** auto-gathering

The solution is that instead of each unit remembering the path or ***FlowField***, each player has two ***FlowFields*** assigned at the start of the game. A player's first ***FlowField*** holds information about the shortest path from each ground tile on the map to the nearest Warehouse owned by the player. The second ***FlowField*** has the shortest path from each ***Water*** tile to the closest Fishery owned by the according player. Both ***FlowFields*** get calculated by *Flow-field* pathfinding and serve for ***AutoGather*** units to navigate back to the deposit building when their inventory fills.

For a player, each player's building is a valid destination for its auto-gathering ***FlowField*** (Figure 3.6). This way, when the map changes, it is necessary to

update only (2 * number of players) a few *FlowFields*, and all auto-gathering paths back are updated.

3.6 Unit information sharing

A game with multiple units usually requires information (like the number of gatherable items from specific resources or following a target) to be shared between the Units. The problem with sharing is the data needs to be processed in real-time swiftly with little overhead because some other actions might depend on the data received (for example, information that the other unit is too close or attacked).

Integrating ECS with Events/Unity Events [24] is one option for passing such information. A nice treat of an event-driven environment (above other implementations) is clean and self-explanatory code. However, the scalability over many units usually leads to an overwhelming number of events triggered at once (for example, spacing between units in large groups), which could be better for its computation/memory requirements. The other reason is the ECS needs to be better adapted to work with Game Objects, which are usually the objects events are bound to/ raised from.

The ECS solves this problem because the data are not bound to an Entity. This way, the data are processed Entity independent. The Systems processing the Entity Components' data do not need to understand all Entity Components, only those the System operates over.

One System can manage all movement over units with scheduled paths (using only read-only access permissions to read a unit's data). The other System can meanwhile run with read-write permissions transforming /recalculating (obsolete) data.

To share information between Systems, the Tag Components [14] are usually used (for example, System for archery does not need to know how to deallocate/ remove an Entity with 0 HP, so it only flags it with the *ToBeDestroyed* Tag Component, and other Systems running over these Tag Components will process this Entity).

The Systems can run every frame or only when an EntityQuery [23] is satisfied. The EntityQuery is essential for working with Systems because we can mark which data and with which permissions (*ReadOnly* vs. *ReadWrite*) we want to get from Entities that satisfy the EntityQuery. The EntityQuery can also specify Entities entering a Job in the Unity Job System.

The Jobs then iterates only over carefully selected Entities, fully embracing the power of multithreaded Jobyfied code with Burst [4]. The main drawback is that Jobs work only over unmanaged data, and only allowed collections (inevitable for pathfinding) within a Job are the Native collections.

The native collections [57] present a thread-save unmanaged alternative to the managed C# collections. The main difference is the necessity of deallocating them manually on Job completion. The second difference is they do not support multidimensional indexing (e.g., multidimensional arrays). We have solved this issue by flattening the collections and indexing them with a flat index instead.

4. User's documentation

The game lore is that we, amongst other players, appear on islands and form a team with survivors. It is essential to stay alive for the longest time to be the Last clan, whether via letting our opponent starve or attacking him to fasten the victory. Each player controls one clan, and every clan belongs to one of two teams with a maximum of 3 players/clans in each team.

During one game, a player gathers resources to build ships, Warehouses, Fisheries, or recruit units. Ships gather food resources on the seas (*fishing*), where the land units can gather minerals and cut trees. Every gathering unit, whether *fishing* or *tree-cutting*, must deposit the resources to the according deposit building to add these resources from the unit's inventory to the players (the one who owns these units).

All land units need to feed one food resource periodically. With feeding comes another mechanic: *starving*. Starving is a game mechanic where a random (land) unit gets destroyed when a player who owns them does not possess food to feed them.

The game goal is to stay alive for the longest, or at least longer than the enemies. We can achieve this directly via fighting or indirectly by defending to the point where the enemy starves to death. The game ends when all the clans' land units die/the enemy team dies, leaving the survivor team as winners.

Extended controls are mentioned in the Table of controls appendix A.

4.1 Setting up a game

To start the program, run **`TheLastClan.exe`** which can be found on the attached CD B. The game initially welcomes a player with a menu where the player can choose a new game or enter the map editor.

After selecting the new game, the player is presented (Figure 6) with numerous options about the game, such as resources that each player begins with, map selection, or team size.

After setting the options, we can click the Start the game button, and the game starts launching.

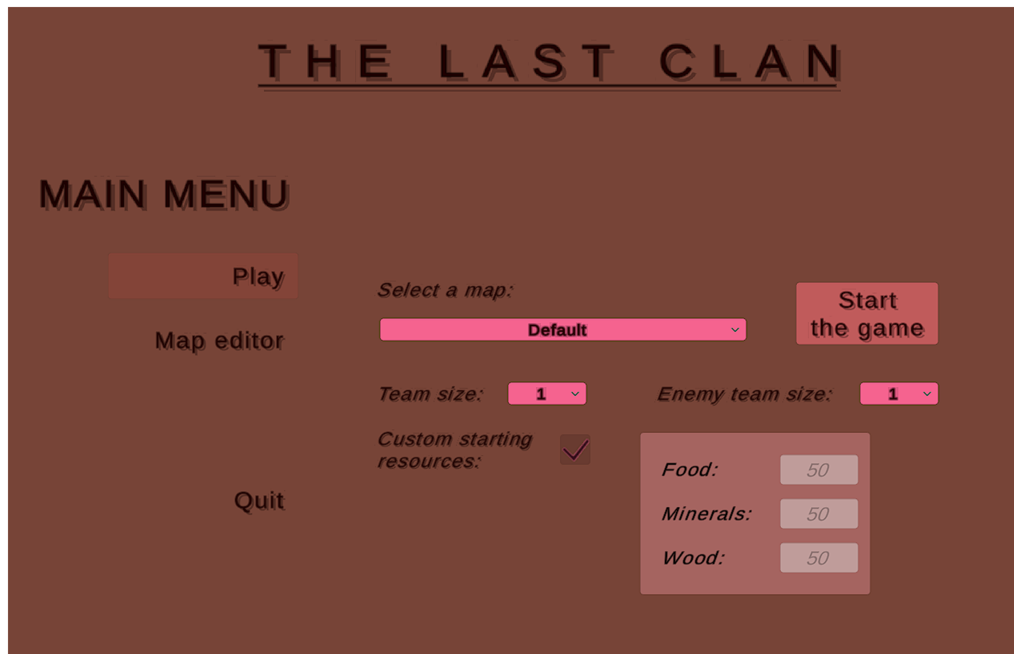

Figure 4.1: *Menu* Scene shows the game settings after the user clicks on the Play button.

4.2 Gameplay

We are spawned, as the game starts, with initial resources two land units, a Fishery, and a ship. Every game Entity can be selected (**left mouse button**) to show additional information (Figure 4.2) about the unit, building, or resource.

The user interface consists of three main parts: total resources indicator, mini-map, and the selected Entity info.

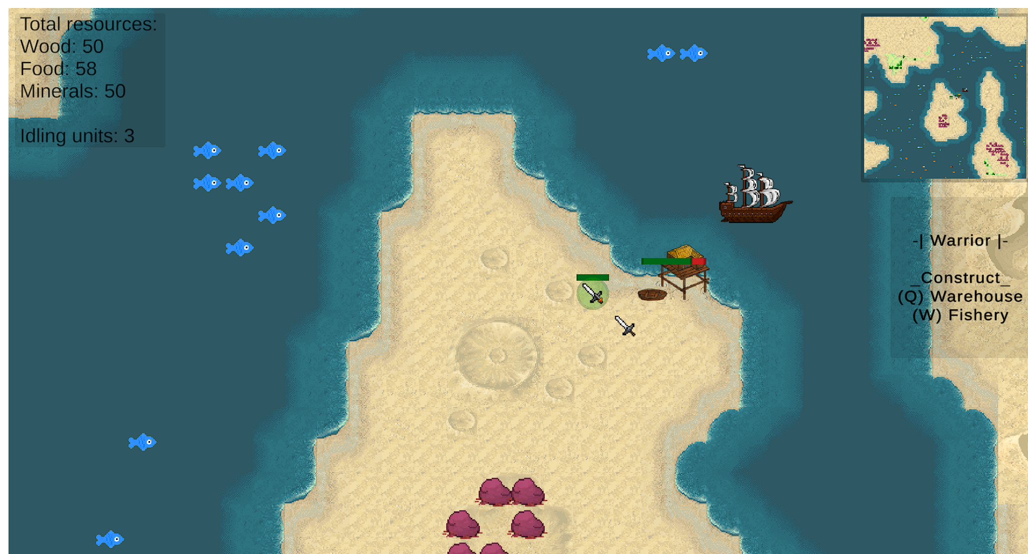

Figure 4.2: User interface after a unit gets selected.

Total resources, shown in the upper left corner of the interface, holds information about the player's inventory plus the number of idling units. The selected

unit info works similarly, located under the mini-map showing info about the currently selected object (requires one).

The next step of the game is to start gathering to become self-sufficient without worrying about the food resource obtainable from fishing. If no food resources are available in the player's inventory, the units begin to die one by one randomly, and the starving warning appears (Figure 4.3). To prevent more units from dying, gather more food, which also turns off this warning.

Figure 4.3: A starving warning message appears if the player's food supplies are empty.

To begin fishing, right-click on the fishing spot with the ship selected, and the ship automatically starts migrating between fishing spots and Fishery to deposit filled inventory (Figure 4.4). Besides serving as the deposit building for food, the Fishery enables players to build more ships.

Figure 4.4: A ship sent to fish on a fishing spot will flow to the spot and start fishing automatically if the spot is nearby.

To start gathering on *land*, first, we must select a unit and build Warehouse. The Warehouse can recruit new units or deposit the land resources to the player's inventory. To construct the Warehouse or the Fishery, we need enough materials for the building construction, select a controllable unit and press the **Q** or the **W** key, hovering with a mouse over the area where the new building should place.

Every building stands on 2x2 tiles and has to stand on specific type tiles. The Warehouse can only stand if all four tiles are *land* tiles, whereas the Fishery has to stand on at least one *water* and *land* tile.

However, the units cannot build over the *rough terrain* and *mineral deposit* or a warning message of the inability to place buildings briefly shows. The warning also shows when there are not enough resources for the building construction.

The *rough terrain* also slows units passing over it, as opposed to the *grass* positively impacting unit movement.

If no units, resources, or buildings get targeted by the **right mouse button**, the standard ***MoveTo*** operation is performed over the selected units.

After building the first Warehouse, the units can deposit gathered materials there, and the *auto-gathering* feature is on. The *auto-gathering* means the selected unit starts migrating back and forth (Figure 4.5) from resource to the Warehouse like a ship is migrating between fishing spots and Fishery.

Figure 4.5: *Auto-gathering* units will automatically return to the nearest deposit building to empty their inventories if full (arrows added for clarity). The units will also automatically search for the nearest resource if their lastly selected got depleted.

With enough resources secured player can start attacking. The attacking works similarly to other actions: select owned units and **right-click** on the target (Figure 4.6). After the target gets to 0 health points, it automatically disappears. The units capable of attacking are warriors and archers, each having a different attack range, damage, and health points.

Figure 4.6: Archers and Warriors can fight buildings or *land* units. The unit marked to attack enemies will damage them if in attack range.

4.3 Map editor

The map editor is accessible via the menu. The player clicks the map editor button and loads an empty 200x200 tile customizable map.

There are a total number of 5 map surfaces, where each has different properties. The *grass* is a spawning area for trees and the *mineral deposits* for minerals. The resources get to spawn automatically. The map editor only creates the possible spawn locations.

To load and save the map, fill the map name textbox (with the "Enter the map name..." placeholder) and click on the load / save button with the map name textbox filled. The maps are saved and loaded in a text format from the system-dependent permanent storage path.

The map can be edited by a mouse cursor (**left click**) working like a brush using the selected surfaces as paint and the Tilemap as a canvas. There is also an option to fill a selected area with the currently selected brush if holding the **left Control** key, as Figure 4.7 shows.

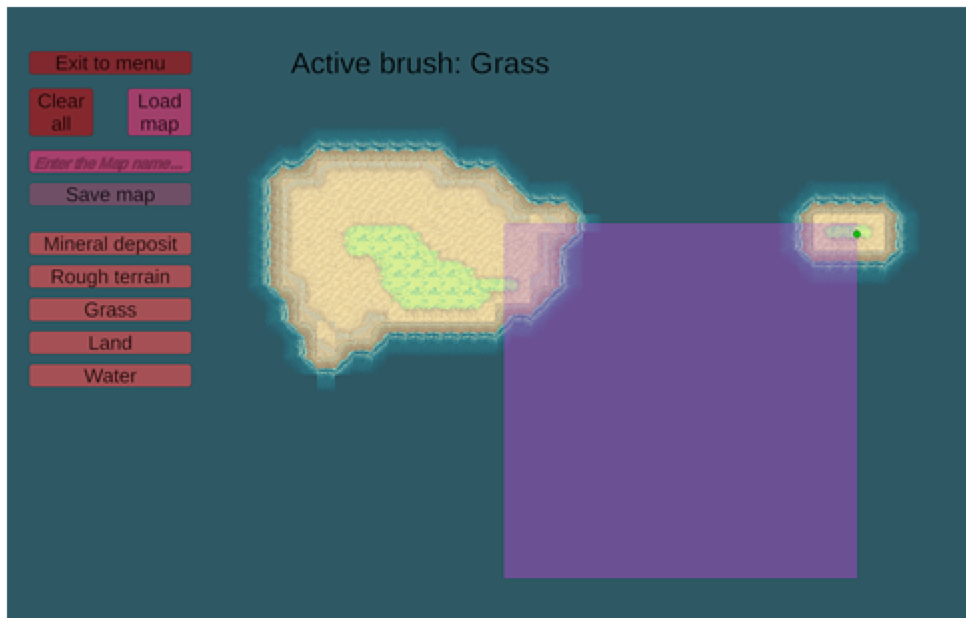

Figure 4.7: Map editor

5. Programmer's documentation

We can separate the development of our game into design and logic. The scripts determine the game logic since they operate on the game data and transform them. The graphics can seem less important; choosing poor graphical design or technology can have negative performance implications, even if scripts run optimally.

In our project, the implementation lies in the **/Assets** directory, where the main game logic is in the **/Prefabs**, **/Scenes**, and **/Scripts** directory.

The **/Prefabs** contain instantiable Game Objects later used to spawn ECS Entities, where the **/Scenes** directory holds *Game*, *Menu* and *MapEditor* Scenes. The other directories contain mostly graphical Assets like Game Object materials for hp and selection circle rendering, palettes for drawing into the Tilemap, and others: sprites, fonts, and tiles.

The **/Scripts** directory 5.1 holds **/Components**, **/Systems**, MonoBehaviour, and Scene management scripts **/MapEditor/MapEditorController.cs** and **/Menu/MainMenuController.cs**.

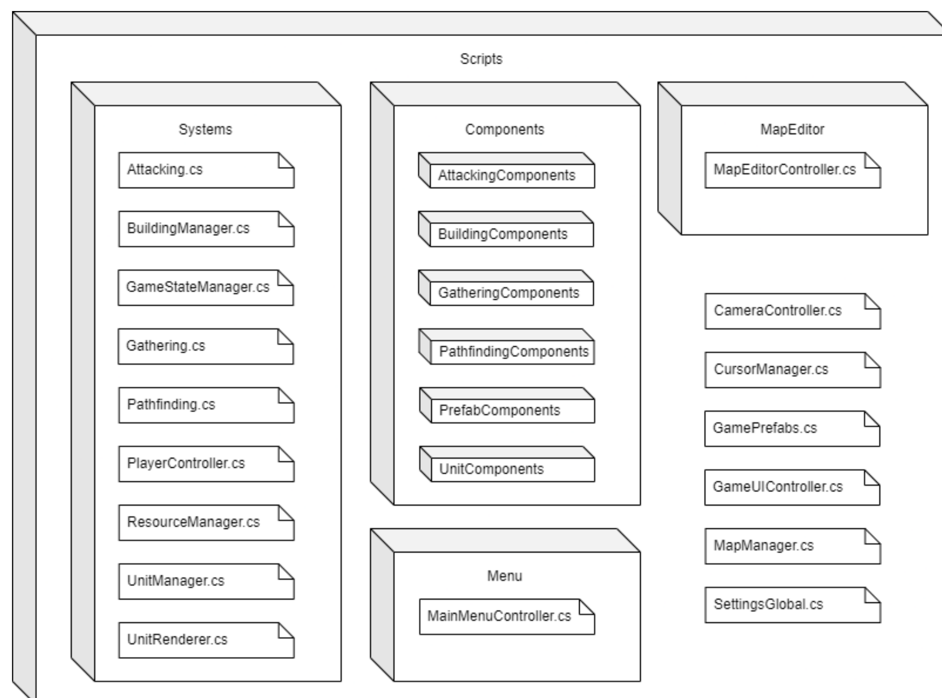

Figure 5.1: Structure of the **/Scripts** directory containing 6 Scripts and 4 directories. The directories are: **/Systems**, **/Components**, **/Menu** and **/MapEditor**.

The **/Component** directory includes (ECS) Components representing properties of all in-game Entities.

The MonoBehaviour scripts are scripts inheriting from the MonoBehaviour base class [36], capable of interacting and transforming the Game Objects. Examples of MonoBehaviour scrips in my projects are **CameraController.cs** enabling the player to control the camera or **MapManager.cs** holding/managing information about the current map environment (tied to Tilemap), like size or depth map.

The scripts in **/Systems** directory are Systems, the most important in the game. Systems inherit from the ***SystemBase*** a base class from ECS, enabling them to manage Entities optimally with the Job System and Burst compilation support.

Some scripts can have the *Controller* or the *Manager* suffix. The controllers are usually scripts directly/indirectly working with user input, whereas the managers are Systems transforming Entities' data and managing them. Examples might be **GameStateManager.cs** or **PlayerController.cs**.

5.1 Environment preparation

The environment we have chosen is Unity 2020.3.16fl, distributed under the unpaid license for projects not exceeding a certain amount of money gained from the game. The Unity version is available via the Unity Hub managing all Unity versions installed on the machines for all added projects.

The main package that needs installing via the package manager is Unity Entities [21], which is part of the DOTS. The DOTS also requires additional packages, nowadays automatically installed with the Entity package. Required (automatically downloaded) packages are Hybrid Renderer, Jobs, Netcode, Collections, Mathematics, and Burst.

The Hybrid renderer [28] is a System managing the rendering data of all Entities and sending these data to Unity's existing rendering architecture. The Collections [57] contain unmanaged Native collections required while working with Jobs on multithreaded code. The Jobs [30] extends the unity core Job System over the DOTS package. Mathematics [35] provides functions necessary for optimal working with the Burst package [4], compiling C# to highly efficient native code.

The game also uses the TextMeshPro package [35], essential for creating UI with text that is not blurry and fully customizable.

All the 2D graphics used in the game is under CC (Creative Commons) distributable and adaptable licenses from Opengameart.org [38], where the file with all resources is in the **/Assets/Sprites/credits.md** file attached to the project.

5.2 The Game Scene lifecycle

When starting the ***Game*** Scene, which is the primary Scene in the project, several Systems run in a particular order and then continue to loop until the end of the game.

Everything begins in the ***Menu*** Scene screen 5.2, where the player selects a map plus additional options to the game and clicks on the Start the game button.

First, the Prefabs with ***ConvertToEntity*** Components get converted into Entities, essential for later instantiating new Entities from these Prefabs. The Prefabs are initially in the ***Game*** Scene hierarchy under the Prefab Game Object.

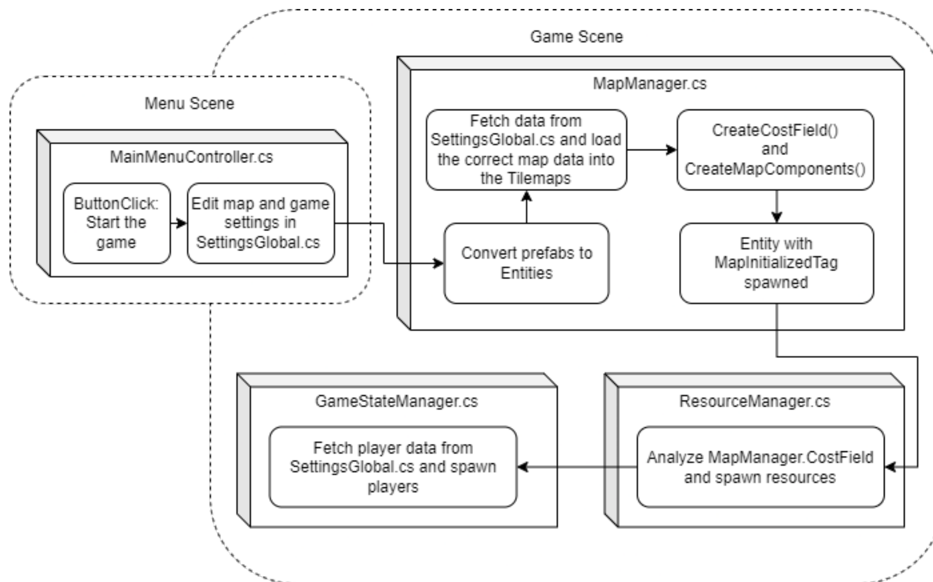

Figure 5.2: User input from the **MapManagerMenu.cs** in *Menu* Scene initiates the start of the game, switching Scene to the *Game* Scene where the map initialization starts. The map gets initialized in **MapManager.cs**, and resources and players get spawned by the **ResourceManager.cs** and the **GameStateManager.cs**.

The next step **MapManager.cs**: checks the static class **SettingsGlobal.cs** holding information about the selected map, resources, and other initial settings. The **MapManager.cs** then redraws Tilemaps (the most time-consuming part): according to the selected map in the *Menu* Scene.

As soon as the Tilemap instantiates, the **MapManager.cs** runs *CreateCostField()*, *CreateMapComponents()*, *CreateMapComponentAdjacencyMatrix() CreateComponentDepthMap()* methods, essential for pathfinding, building, and resource spawning.

After successfully initializing the map, **MapManager.cs** spawns an Entity with *MapInitializedTag*, signalizing the **ResourceManager.cs** and the **GameStateManager.cs** to run.

The **ResourceManager.cs** starts first, analyzes the *CostField* and runs *SpawnTrees()*, *SpawnFishingSpots()*, and *SpawnMinerals()* methods spawning all the resources available on the map during the game.

The **GameStateManager.cs** checks **SettingsGlobal.cs**, calculates the spawn positions of all players, and then spawns them with the correct: resources/units.

At the end of the initialization phase, other essential Systems have their loop: the **Attacking.cs**, the **Gathering.cs**, and **Pathfinding.cs** (amongst other smaller Systems), transforming units, paths, targets, and resources. All are running in a loop, searching if any EntityQueries matching Entities need to be processed.

Another notable System loop might be the **UnitManager.cs** running an EntityQuery over all unit Entities and checking if any is starving/idle. Or the loop of **GameStateManager.cs** checking if the game ended already and who won.

5.3 Map creating

We have created the map using Tilemap. The Tilemaps are used both in the Game (Figure 5.3) and *MapEditor* Scene. When creating a Tilemap, multiple grids can be instantiated under one object creating a multi-layered Tilemap. There are six grids: each used to depict different map layers. The ***Water*** and ***Land*** are disjunct, where the ***Grass***, ***MineralDeposit*** and ***RoughTerrain*** can be placed only on the ***Land***.

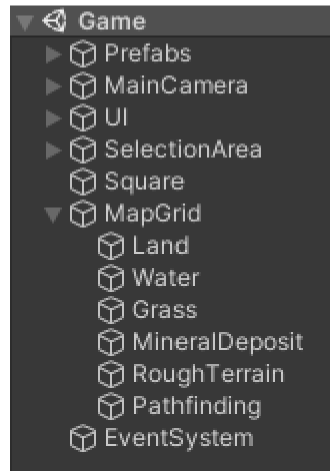

Figure 5.3: The Game Objects hierarchy in the ***Game*** Scene. The ***Game*** Scene contains: ***Prefabs***, cameras, UI elements, and ***MapGrid*** holding 6 Tilemaps.

Splitting into multiple Tilemaps layers can significantly ease the work done on the map editor because all Tilemaps have building methods like ***HasTile (Vecor3Int position)***, or ***SetTile(Vector3Int position, Tilemaps.TileBase)***. The ***HasTile()*** returns true if a tile was painted/placed on the given location, making the Tilemap well addressable. And the ***SetTile()*** method puts a Prefab tile in the given position. Thus, detecting where the ***Water*** (obstacle) starts; could be done via ***Water.HasTile()***.

Every Game Object also possesses the rendering layer [32] assigned, which determines which Game Object will render in front and which in the back.

The Tilemaps get edited via brushes in the Scene view. They must all belong to some palette (Figure 5.4). The project has two palettes located in the **/Assets/Palettes** directory. The **ground_palette** is for map design, and the **pathfinding_pallete** is for drawing arrows into the ***Pathfinding*** Tilemap.

Tiles added to the pallet can have many types, where some work like a simple Sprite (picture), spawned on the brush tip position, while others might have deeper functionality. The others might be animated tiles switching the Sprites in a given sequence. Still, for this project, we have chosen the Rule tiles [45] because they combine simple and animated tiles and add logic to the tile rendering.

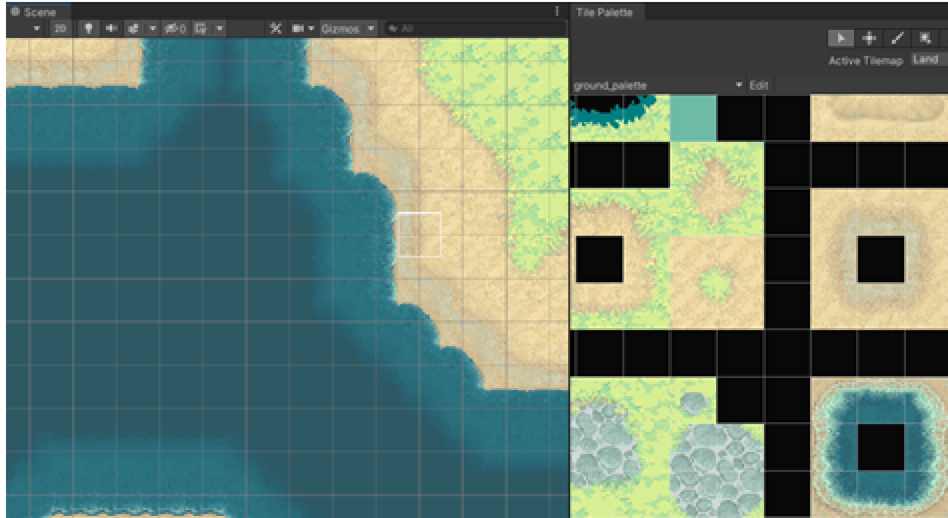

Figure 5.4: Tilemap editor on the left and the Tile palette ***ground_palette*** on the right.

The Rule tiles can react to neighbor tiles of the same type and transform accordingly (Figure 5.5), giving an interactive feeling to the user.

Figure 5.5: ***Land*** Rule tile gets defined by a set of rules. Each rule is described by a 3x3 grid representing the tiles around. Different markings in the grid represent different constraints: empty space, cross, or arrow corresponds to the position/edge where the tile can, can not, or should be connected to the other neighbor tiles.

The main advantage of using brushes and tiles is that the scripts depend only on the Tilemap bound to them, meaning we can freely change the rules of tiles, graphics, or frames on animated tiles.

The other great benefit is the simplicity of rewriting a map from a file (loading a custom map), affecting only a few Tilemaps, without the necessity of spawning multiple individual Game Objects and managing them. The preferred design is to keep fewer Game Objects in the solution since the workflow between ECS and Game Objects still works poorly in the 0.51 Entities. Or at least split the Mono scripts and the *SystemBase* scripts.

The **MapManager.cs** is the main script editing the Tilemaps with direct reference to this Tilemap Game Object, which is possible because the **MapManager.cs** class inherits from MonoBehaviour. It also redraws the map according to the map in **SettingsGlobal.cs** using a similar technique as placing tiles in the **MapEditor.cs** (*HasTile()*, *PlaceTile()*).

One of the essential methods of **MapManager.cs** is *CreateMapComponents()* which separates the map of the components and saves the component of each tile to the corresponding x,y position in *int Components[,]*.

Figure 5.6: An example of the possible numbering of components in *Components[,]* generated by the *CreateMapComponents()* method; both implemented in the *MapManager.cs*.

The **MapManager.cs** also counts the depth map *ComponentDepthMap* used to select and find the correct spawn points for *FishingSpots* and buildings. It also manages the *CostField*, which holds information about all obstacles in the game. The primary usability of *CostField* is for pathfinding and building.

Other objects presented in the game as trees, minerals, units, buildings, or fishing spots are spawned later and do not affect the Tilemap (only the *CostField*).

5.4 User Interface

The first essential thing to solve in RTS in the user interface is controlling the camera. A camera [5] is a Game Object enabling a player to see the game world

from the position where the camera currently is. The *Game* has two cameras: the *MainCamera* and the *MinimapCamera* (used for the mini-map).

The camera controls are located in **/Assets/Scripts** directory in **Camera-Controller.cs**, where the script *Transforms* Components of the camera Game Object to match the user input. The script edits the *Transform* Component of the *MainCamera* to make it move (panning).

The *MinimapCamera*, on the other hand, is indirectly controlled by the user because we need the mini-map camera to always be in the same position as the *MainCamera*, only a bit further (zoomed out).

In Unity, all Game Objects have relative positions linked to their parents. The parent-relative positioning offers a solution for the *MinimapCamera* position where we the *MinimapCamera* Game Object under the *MainCamera* object in the Scene hierarchy. Then every change of the *MainCamera*'s Transform Component propagates to the *MinimapCamera* since its position is relative to the parent Game Object. The *MinimapCamera* then sends its output to the UI element, which renders the final mini-map in the top right corner.

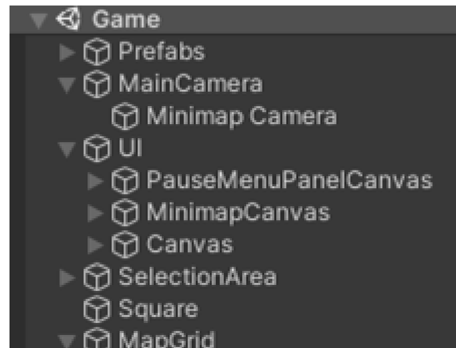

Figure 5.7: *MainCamera* is a parent Game Object to the *MinimapCamera* in the *Game* Scene hierarchy.

The alternative to using a standard camera Game Object is Cinemachine [6], which is better for different game genres since it offers many more methods to pan the camera or bind it to a character.

The problem was defining the map borders. In the classic Camera solution, we are just transforming the data and checking if the camera (even if zoomed out) is still in the map boundaries. But Cinemachine would require world borders to be defined, which is another Game Object that can affect physics, and in my opinion, it was just not necessary to use it.

All the UI elements need a canvas parent, which works like a container for the UI elements. Showing and hiding UI information then falls into turning Game Objects on and off.

We are also using the TextMeshPro package [52], which offers enhanced UI elements with more than just basic text editing and without blurry edges.

The other two mentionable UI elements in the *Game* Scene are the total resources indicator and the current unit info. Both are TextMeshPro textboxes with panels hidden behind them. The **GameUIController.cs** then assign the data to the textbox; it also manages almost all UI elements in the *Game* Scene, like showing the UI error messages.

The **GameUIController.cs** inherit from the MonoBehaviour, which means it can edit objects (as he does with the UI elements); however, the problem is that Entities which are the data from being accessed by classes implementing the *SystemBase* and not the MonoBehaviour (required to manage Game Objects) scripts. However, the MonoBehaviour script can use the reference on the Entity manager via *World.DefaultGameObjectInjectionWorld.EntityManager*, which is always present in the World. Using this approach, we can query Entity data without intervention with any Systems like **ResourceManager.cs** if we want to know more resource info.

When a user clicks on an Entity, an EntityQuery is crafted over all selected Entities via the *DrawSelected()* method to access the Entity info. The **GameUIController.cs** usually seek selected units marked by the **UnitManagerSystem.cs** with the *SelectedTag* minimizing the data required to fetch from the ECS via MonoBehaviour script to Game Object like a text box.

5.5 Menu and Map editor

The menu uses similar objects, like the ones mentioned in the User interface, but on top of loads, Scenes. The main script controlling menu is the **MainMenuController.cs** located in the **/Asset/Scripts/Menu** directory.

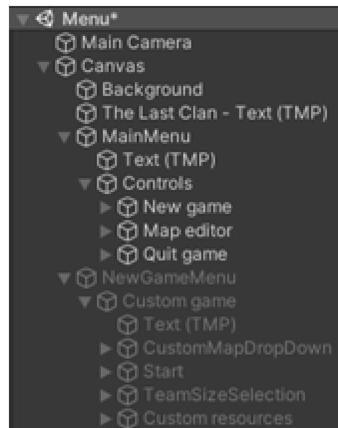

Figure 5.8: Menu hierarchy

The two primary tasks *Menu* handles are switching between Scenes and interacting with the **SettingsGlobal.cs** static class from which necessary data for launching the *Game* Scene loads. Scenes present in the solution are the *Game*, *Menu* and *MapEditor*, where the *Menu* Scene loads them based on the user's input. The *Menu* also searches in the map folder for any custom maps available.

The map editor is the one adding custom maps to the map directory. It has a Tilemaps structure like the Game Scene, but editable. It uses the *HasTile* and *PlaceTile* mentioned earlier and saves it to persistent storage [40]. Each tile is represented by one char when saving and loading tiles, as Figure 5.9 shows.

Figure 5.9: Map text file representing **W** character for *`Water`* tiles, **L** for *`Land`* tiles, **R** for *`RoughTerrain`* tiles, the **G** for *`Grass`* tiles, and **R** for *`MineralDeposit`* tiles.

5.6 ECS Entities and Components

The ECS introduces Entities, Components, and Systems. The Entities have Components, and the Systems transform the data in these Components. To fully utilize the ECS technology, it is a good practice to split Systems to work only over specific Components because then the only System required to have ReadWrite data access is the one editing it, and others can benefit from the multithreading while only reading the transformed data. An example of such a System might be **`ResourceManager.cs`**, which only manages the Entities with resource-specific Tag Components and does not need to understand the Components for unit gathering (implemented in **`Gathering.cs`**).

A unique combination of Components is called an Archetype representing a specific unit type (all archers fall into the same Archetype). The Systems can then run effectively over the data they need.

A Component is a structure (or class but not recommended) inheriting from the *`IComponentData`* containing variables of unmanaged type example in the project is **`Health.cs`** or **`Damage.cs`**.

Other Components used are the Dynamic Buffer Components [10] that work as resizable arrays over unmanaged types, mostly usable for pathfinding, or the Tag Components, which are empty structures inheriting from the *`IComponentData`* serving primarily for Tagging the Entities.

An example of a Tag Component might be the *`ToBeResolvedTag`*, added to yet unresolved pathfinding target.

The Components are in the **`/Assets/Scripts/Components`** directory.

The essential Archetypes in the project represent units, tasks, buildings, and resources.

5.6.1 Archetypes and instancing Entities

The game contains multiple Archetypes and Systems running over their Components, which allows us to specify EntityQueries selecting only the data needed for a specific Job.

Every Archetype in the program could split into two categories: having System specific or independent Components.

System-specific Components are those only one System edits or uses. The System-specific Components are in the chapter dedicated to the according System.

The important Archetypes containing most System-independent Components are units like *Archer* (Figure 5.10), *Ship*, and *Warrior*.

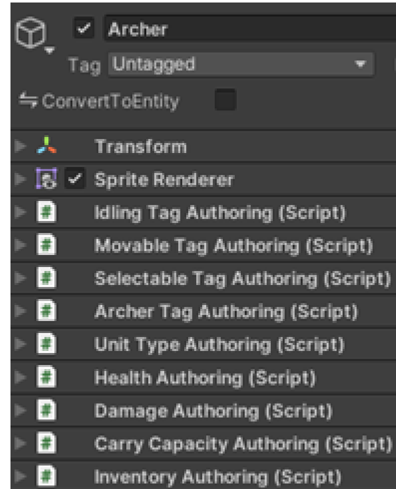

Figure 5.10: *Archer* Prefab, used for Entity instancing, attached with Component Scripts like *ArcherTag*, *MovableTag* or *UnitType*.

To instantiate Entities with the same properties (like buildings, units, and resources), we specify a "Prefab" Archetype that works like a Prefab.

In the standard Game Object-oriented design, the Game Objects instantiate from Prefabs. Still, since the Entities, unlike the Game Object, do not support the MonoBehaviour scripts (scripts assignable to Game Object/Prefab), the ECS introduces the conversion workflow from Game Objects to Entities. Prefabs are convertible in runtime via the *ConverToEntity* script (Figure 5.11). We are then able to instantiate from this converted Entity. Examples of instancing from such Prefab are *SpawnUnit()* method in the **PlayerController.cs** or *SpawnPrefab-Building()* in **BuildingManager.cs**.

The other benefit is that we can instantiate data with reference types (Meshes, sprites) which would be otherwise problematic in the Systems running mostly over unmanaged data.

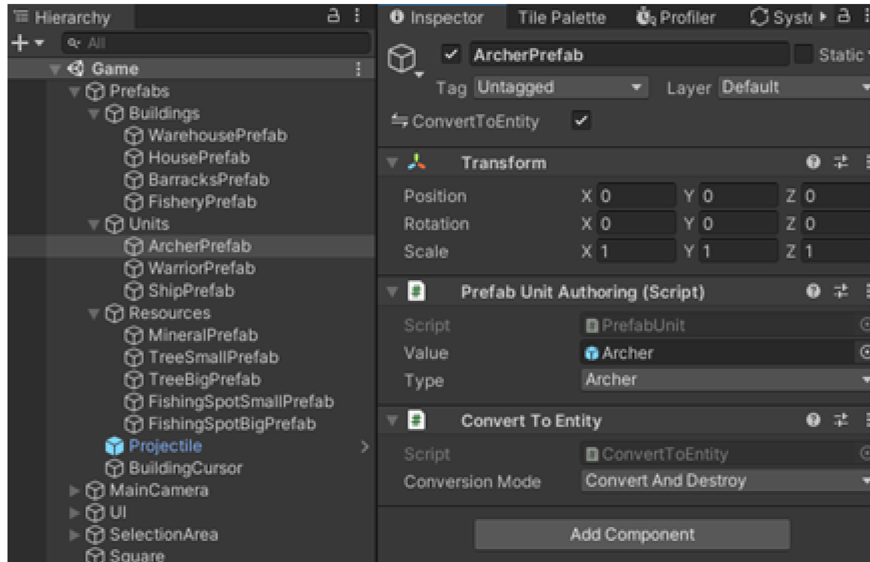

Figure 5.11: *ArcherPrefab* detail, located in the *Game* Scene Prefabs hierarchy. The *Game* Scene holds all Prefabs as children under the *Prefabs* Game Object, which further separates into *Buildings*, *Units*, *Resources* parent Game Objects of individual Prefabs. These Prefabs get converted into Entities on Scene load and will be used to instantiate new buildings, units, and resources.

The reason why are Game Objects (Prefabs) able to have ECS Components (implementing *IComponentData*) in them is by marking the structures as *[GenerateAuthoringComponent]* which is then allowed to be associated with a Game Object. All Prefabs are in the **/Assets/Prefab** directory.

The conversion workflow is solved better in Entities 1.0, where the Baker workflow [2] is used.

5.7 Entity Component Systems

Systems are scripts giving logic to the ECS System by transforming the Entities' Components. Essential Systems are **Attacking.cs**, **Pathfinding.cs**, **Gathering.cs**, **ResourceManager.cs**, **UnitManager.cs**, **BuildingManager.cs**, **PlayerController.cs** and the **GameStateManager.cs**. All of these are in the **/Assets/Scripts/Systems** directory.

Each System manages different functionality, usually containing one or more Jobs. The usual structure of a System is the handler method preparing an EntityQuery and the data necessary for a Job. After fetching all the data, the handler method schedules the Job and deallocates the data on Job completion. The EntityQuery is essential for specifying which Components/Archetypes the Job needs - meaning if there is no Component and the EntityQuery is empty, the System/Job will not run.

The Jobs used by the Systems usually work with Native collections that do not support multidimensional addressing and need manual deallocation. We can, however, create a one-dimensional array indexed by the (one) flat integer index to act like multidimensional collection-like structures.

Let us have a 2D *array[arraySize.x, arraySize.y]*, the flattened index of its one-dimensional version would look like this:

```
int FlatIndex(Vector2Int index, arraySize.x){
    return arraySize.x * index.x + index.y;
}
```

with conversion back to the 2D index:

```
Vector2Int index = new Vector2Int {
    y = flatIndex / arraySize.x,
    x = flatIndex % arraySize.x
};
```

The **Attacking.cs**, **Pathfinding.cs**, **Gathering.cs**, **UnitManager.cs** and **PlayerController.cs** are Systems directly affecting units meaning they edit unit data based on the Tags found on them.

Where the **ResourceManager.cs**, **BuildingManager.cs** and **GameStateManager.cs** usually do not target units specifically.

To take a look from a player's perspective. The player selects units and schedules a task for them. The **PlayerController.cs** manage to assign the correct Components for the action chosen.

For example, if the action is a move order, the *MoveTo* Component is added to the selected Entity. The **Pathfinding.cs** then calculates the moving path and move the Entity. Other Systems might get active while working on these data like the **UnitManager.cs** reacting to all current Components associated with a unit and removing the *IdlingTag* Component

5.7.1 Pathfinding System

The *Pathfinding* System, implemented in **Pathfinding.cs**, manages path calculation for all units in the game and their movement.

The main methods of the System are: *WaterMovement()*, *GroundMovement()*, *UnitSpacing()*, *CalculateWaterPaths()*, *CalculateLandPaths()* and *RemoveUnusedTargets()*.

The main Components are *Target*, *ToBeResolvedTag*, *FloatingTargetTag*, *FlowFieldCell*, *MoveTo*, *Movable*, and *SailStartPoint*. Most are in the **/Assets/Scripts/Components/PathfindingComponents** directory.

The *Pathfinding* System workflow usually works as follows: a movable Entity is selected, and the moving action to chosen direction is required. Then, an Entity with a *Target* Component (with the desired direction assigned) and the *ToBeResolvedTag* (optionally the *FloatingTargetTag* if ship selected) spawn. "Reference" to that target Entity is added to the *MoveTo* Component assigned to the selected unit. The benefit of using this design is the other Jobs can see if a unit is ready to be moved - having the *ToBeResolvedTag* Component on its target Entity.

The *GroundMovement()* and *WaterMovement()* methods are both handler methods fetching data and creating EntityQueries for the same *MoveUnitJob*, which is another example of the reusability of the ECS code via EntityQueries. Having two handler methods then allows us to run the Job with different parameters like the movement speed (different for *Land* and *Water*).

The *UnitSpacing()* method manages forces between units and units and obstacles and units, which means when a unit overlaps with another unit or building, the unit gets pushed away. The forces between ships are significantly weaker because ships use A* pathfinding, as opposed to the *Land FlowFields*, and would be otherwise able to be pushed away from the precalculated array.

To prevent units from being stuck if pushed by *UnitSpacing()* method to a component they cannot move on, the **Pathfinding.cs** implements the *InitializeComponentPushOutFlowField()* method, which initializes the *ComponentPushOutFlowField*. This FlowField contains the shortest path from every component to the nearest and is used to push out every unit standing on the component it should not suppose to.

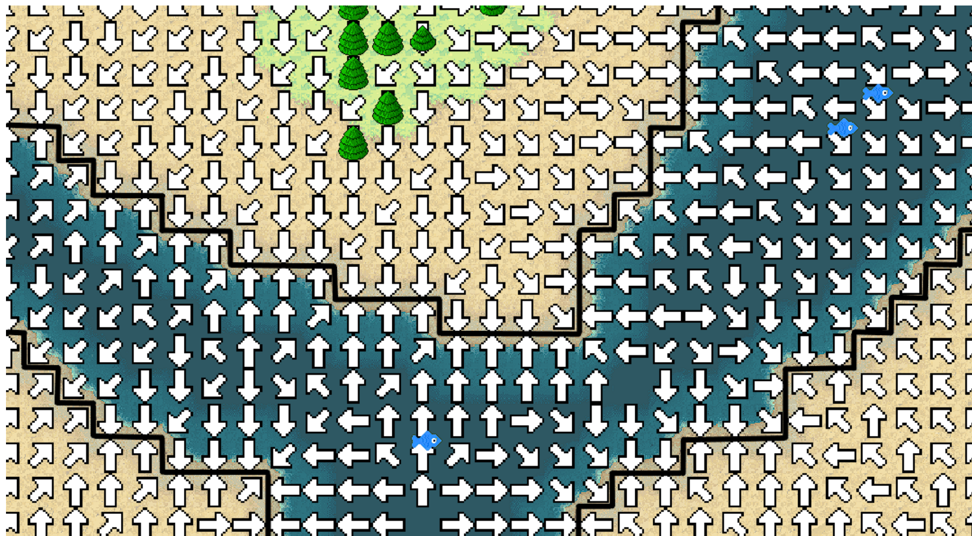

Figure 5.12: *ComponentPushOutFlowField* for pushing units out of invalid components.

The essential methods of the *Pathfinding* Systems are *CalculateLand-Paths()* and *CalculateWaterPaths()*, calculating all paths in the game.

The *CalculateLandPaths()* runs for all *Land* paths marked with the *ToBe-ResolvedTag* and creates *FlowField* using the *Flow/vector-field* pathfinding. The calculation begins with fetching the *CostField* from **MapManger.cs** containing the costs of all tiles in the game (cost of passing a tile meaning impassable walls have the max value). The Job creates an *CostField* from the *Integration-Field*.

The *IntegrationField* (Figure 5.13) has the size of the *CostField*, which has the same 200x200 size as the map. It gets initialized with zero on the target tile position, and the tile index enqueues the priority queue *cellsToVisit*, which always processes the tile with the lowest value. In each step, we dequeue one value

from the queue and check its neighbors' (8 neighbors in total) current integration cost and update its value if the old was lower than the sum of the current tile ***IntegrationField***, ***CostField[neighbor]*** and the distance(current tile, neighbor tile). The distance function uses the Euclidean distance (but Manhattan distance could be in our 2D tile-built world also viable). If the neighbor is updated, its index gets added to the ***cellsToVisit*** queue.

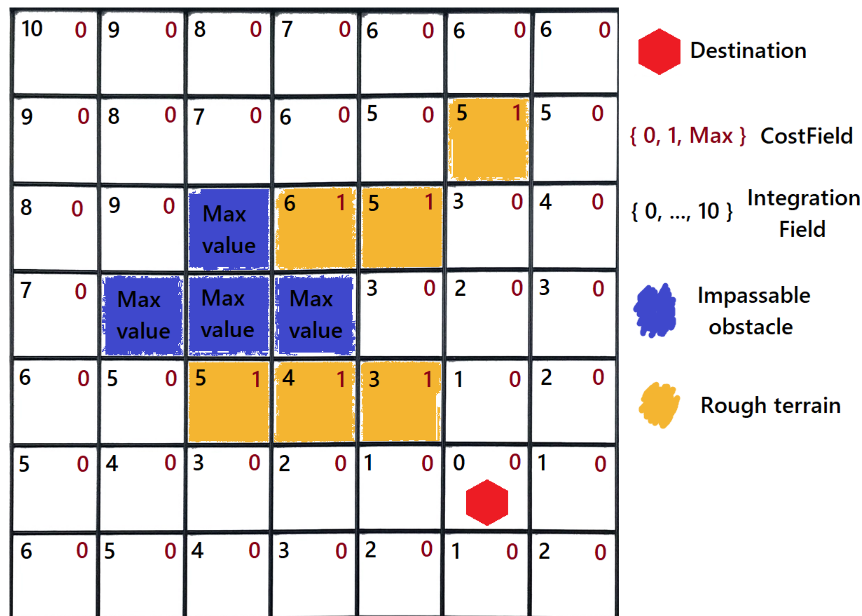

Figure 5.13: *Flow-field* pathfinding, ***Cost & Integration Fields***

The ***FlowField*** gets created from the ***IntegrationField*** with vectors on each tile directing to the neighbor with the lowest ***IntegrationField*** cost (Figure 5.14).

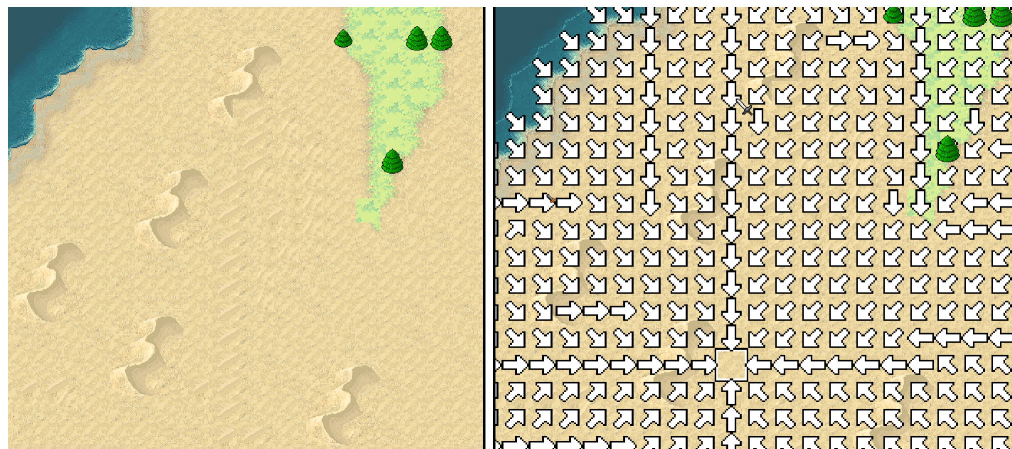

Figure 5.14: In-game example of ***Land*** terrain (on the left) and the ***FlowField*** calculated for the ***Land*** pathfinding over that terrain (on the right).

The ***FlowField*** is a flattened array, where each index contains ***FlowField-Cell***, a Dynamic Buffer Component. This ***FlowField*** gets assigned to a "target" Entity holding the ***Target*** Component, and the ***ToBeResolvedTag*** gets removed.

Then all units with ***MoveTo*** Component pointing to this "target" Entity are moved by the ***MoveLandUnits()*** method because their target got resolved. The target Entity gets resolved when no ***ToBeResolvedTag*** Component is associated with it. The position of a unit determines the index of the ***FlowFieldCell*** Component held in the target Entity, which corresponds to the movement vector the unit need to follow.

The *Flow-field* algorithm can also find the closest deposit building for all ***AutoGatherTag*** units. The auto-gathering unit automatically migrates between resource and deposit buildings (Figure 3.6). The algorithm is a slightly edited version of the one mentioned, placing more destination tiles at the initialization of the ***IntegrationField***, where each building position is a destination, whether it is a ***Land*** path to a Warehouse on the ***Water*** to a Fishery. Each player is initially spawned (by the **GameStateManager.cs**) with two permanent ***FlowFields***, one for ***Water*** auto-gathering and the other for ***Land*** auto-gathering, to hold that information. The unit returning from a resource finds the correct path based on the player and gets assigned to it (not to be moved into enemies' Warehouses).

The ***CalculateWaterPaths()***, on the other hand, uses the A* (A star) algorithm for calculating the paths with the ***Target*** Component, ***FloatingTargetTag***. The A* returns the shortest one to the destination as a queue of vertices leading to the target. We take this queue and build a custom ***FlowField*** from it. The reason is that even though the forces between ships are small, we do not want the ***Water*** units to get out of their path, which would get them stuck and require another recalculation.

Figure 5.15: Ship's vector field for ***Water*** pathfinding. This field gets created from the A* generated list of tiles representing the shortest path to the destination. If any tiles from the list have neighboring ***Land*** tiles, the vector forces pushing from that ***Land*** tile get added to the Ship's vector field.

To keep targets up to date: every change to the ***CostField*** affects the ***Land*** targets and marking them with ***ToBeResolvedTag*** via the ***MapCostFieldChangedMarkObsoleteData()*** method in **MapManager.cs**.

The Entities holding target info about the path get destroyed when no more *MoveTo* Components are pointing to these Entities (except the auto-gathering *FlowFields*) defined in the *RemoveUnusedTargets()* method.

5.7.2 Attacking System

The *Attacking* System manages the dealing and receiving damage from units to buildings and units. There are two types of units able to attack: Archers and Warriors. Archer and Warrior have both the *Attack* Component, but their damage and range differ.

The main Components used in the System are *Health*, *Attack*, *Translation* (position), *AttackTarget*, *Projectile*, and *MarkedForAttackTag*.

The *AttackingUpdateEnemies()* and the *AttackingUpdateAttackers()* are the core methods of the *Attacking* System.

When a unit selects another unit or building as an attack target, the *MarkedForAttackTag* gets added to the target Entity, where the attacker the *AttackTarget* Component. *AttackingUpdateEnemies()* runs an EntityQuery for all targets with *MarkedForAttackTag* and searches if any enemy has *AttackTarget* with an Entity reference equal to the Entity under attack. The *MarkedForAttackTag* Entity then removes the value of *Attack* from its health points for every such enemy in the range.

The *AttackingUpdateAttackers()* runs for all attackers with the *AttackTarget* Component and searches if the attacked unit is nearby. The path gets scheduled to the target if no enemy units are nearby. If the target is moving, its path gets copied to the attacker. (The better solution for a target following would probably be an A* with a heuristic.)

If a target *Health* reaches zero, the *AttackingUpdateEnemies()* destroy the target and remove all Components from other Entities that pointed to this target.

The archers work similarly to the Warriors with an exception in *AttackingUpdateAttacker()*, where if the Archer is in range, it spawns a Projectile with the *ToBeResolvedTag* Component since we are unable to instantiate reference type (like a picture of the Projectile) from the Job. The *ManageProjectile()* method spawns a Projectile, replacing the one with *ToBeResolvedTag* outside the Job.

5.7.3 Building System

The *Building* System manages buildings and their construction. The buildings available in the game are Fishery and Warehouse, each serving as the deposit building for specific resources and as a spawn point for building-specific units.

The Fishery is edited by *FisherySpawningShips()* and the *FisheryDepositResources()* method, whereas the Warehouse is by *WarehouseSpawningUnits()* and *WarehouseDepositResources()* methods.

Both resource deposit methods work similarly, one on the *Water* and the other on *Land*. They withdraw resources from the friendly units inventory (if in range) and transfer them into the player's global resources inventory.

The spawning methods are bound to a specific unit type, and each uses different Components for spawning the units. Fishery uses the *FisherySpawningShips* and Warehouse the *FisheryDepositResources* Component.

When a building is requested to spawn a unit, these Components are added to a building with the corresponding number of units to recruit/ construct. Then the timer is started signalizing the production process. The construction/recruitment time differs based on the unit type. The queue gets automatically deleted if the player does not have enough resources to pay with.

The *Building* System also manages the construction of new buildings, which cannot place on top of other obstacles, and *RoughTerrain*. When a unit gets scheduled with a building construction task, the *BuildingSpawnPoint* Component and *MoveTo* Component with the destination of the construction area get assigned to this unit. If the unit is near the construction, the method *ConstructBuildingIfWorkerNearby()* calls the *SpawnPrefabBuilding()* and again checks if the building is constructible. If the building spawn point is valid and the player has enough resources, it gets instantiated from a Prefab.

All buildings get spawned with a fraction of their health points and repair themselves over time (even if damaged).

5.7.4 Gathering System

The *Gathering* is the second most crucial System after the *Pathfinding* because every unit or building requires the resources for construction or recruitment.

To begin with the *Gathering* System, we have to start from the **ResourceManager.cs**, which spawns all the resources during the map initialization according to locations. The Components used by the **ResourceManager.cs** to work with resources are *TreeTag*, *MineralTag*, *FishingSpotTag* and *ResourceTag*. The other important Component is *Inventory* which tells the gatherer which resources are gatherable from the minerals, fishing spots, or trees.

The locations of the resources are determined by the Tilemap under these resources (minerals only spawn on *MineralDeposit*, whereas the trees only spawn on *Grass*). The algorithms for choosing the specific position are not amusing for trees and minerals, but the fishing spot locations are selected more interestingly.

The fishing spots appear in two forms: *FishingSpotBig* and *FishingSpotSmall*, differentiating in the number of resources gatherable. The *FishingSpotBig* is rarer and thus needs to get spawned less frequently. The spawn points are selected using the *ComponentDepthMap*, which is an array (located in the **MapManager.cs**) that counts the number of tiles from shore and assigns each depth (shore tile has depth 1, their neighbors' depth 2, and so on), where the *FishingSpotBig* get spawned from certain depth thus making it "harder" to obtain.

The *Gathering* System uses more System-specific Components: *MarkedForGather*, *GatherResources*, *AutoGatherTag* and the *AutogatherMovingBackTag*.

The ground units gather the minerals and trees, where the ships fish on fishing spots. When a unit gets scheduled for a resource gathering, the resource is assigned the *MarkedForGatherTag* and the unit with the *GatherResource* Component. The unit also automatically moves to the target managed by the

GatheringUpdateGatherers() method.

```
ResourceUpdate() {
    // runs for each resource
    // "this" refers to the current resource

    if (!HasComponent<MarkedForGatherTag>(this))
        return;

    if (this.Resources.Empty())
        this.Destroy();

    gatherers = getNearbyGatherers();
    if (gatherers.Count() == 0)
        RemoveComponent<MarkedForGatherTag>(this)

    foreach (gatherer in gatherers) {
        if (gatherer.GatherResource.Resource != this)
            continue;

        if (this.Resources.Empty())
            RemoveComponent<GatherResource>(gatherer);

        this.Resources -= gatherer.GatherResource.GatherAmount;
    }

    if (this.Resources.Empty())
        this.Destroy();
}
```

GatheringUpdateResources() pseudocode

If a Warehouse is present, the ***AutoGatherTag*** is automatically associated with the unit. It tells the ***Gathering*** System: when a unit's inventory is out of space, it should return to the deposit building. The coming back action gets signalized by the ***AutogatherMovingBackTag***, and the path to the deposit buildings gets assigned according to the player number. When the unit is in the deposit range of the building, the resources are withdrawn automatically from the inventory. If the ***AutogatherMovingBackTag*** is present, the building removes it. And the ***MoveTo*** is again automatically scheduled to the target unit. The behaviour described is a combination of the deposit-to-the-building methods (*WarehouseDepositResources()* and *FisheryDepostiResources()* located in **BuildingManager.cs**) and the *GatheringUpdateWaterGatherers()* / *GatheringUpdateLandGatherers()* methods.

```
GathererUpdate() {
    // runs for each gathering unit
    // "this" refers to the current unit

    if (!HasComponent<GatherResource>(this))
        return;

```

```
8       if (this.Inventory.IsFull()) {
9           RemoveComponent<GatherResource>(this);
10
11          if (!HasComponent<AutoGatherTag>(this))
12              return;
13
14          newResource = ClosestResource(this.GatherResource.Resource);
15
16          if (newResource == null)
17              return;
18
19          this.GatherResource = newResource;
20
21          if (!HasComponent<MoveTo>(this))
22              AddComponent<MoveTo>(this);
23
24          this.MoveTo = this.GatherResource.Position;
25      }
26
27      if (this.GatherResource.InRange()) {
28          RemoveComponent<MoveTo>(this);
29          resourceType = this.GatherResource.ResourceType;
30          resourceAmount = this.GatherResource.GatherAmount;
31          this.AddToInventory(resourceType, resourceAmount);
32      } else if (!HasComponent<MoveTo>(this)) {
33          AddComponent<MoveTo>(this);
34          this.MoveTo = this.GatherResource.Position;
35      }
36  }
```

GatheringUpdateGatherers() pseudocode

ProcessMarkedForDestroyResources() destroys all *MarkedForDestroyTag* resources, correctly deleting all *Gathering* Components of other gatherers pointing to this Entity. And the *RemoveObsoleteAutoGatherTag()* takes care of the resources no one is gathering, marked with *MarkForGatherTag*.

5.7.5 Unit manager

The unit manager affects all units within the *UnitStarvingSet()* running every few seconds, removing one food resource for every ground unit a player owns.

The other method is *UnitIdlingSet()* marking, idling units with the *IdlingTag* if they have no work to do.

The essential method, however, is GiveOrders(), which controls all computer-driven game units. The GiveOrders() calls *GiveOrdersWarehousesRecruitUnits()*, *GiveOrdersFisheryConstructShips()*, *GiveOrdersWarehousesRecruitUnits()*, *GiveOrdersFisheryConstructShips()*, *GiveOrdersBuilders()*, and *GiveOrdersDefendNearby()*.

The *GiveOrdersWarehousesRecruitUnits()* and *GiveOrdersFisheryConstructShips()* methods are responsible for the computer-driven production of more units. However, the strategy of how many units, which type, and in what warehouse should recruit the units is strictly random.

The *GiveOrdersGatherersWater()* and *GiveOrdersGatherersLand()* give units orders to gather on the same component as the unit's position since every unit in the game has limited movement to only the component they have got spawned on. Whenever a new idle and non-player unit appears in the game, it gets assigned a gathering task. Each computer-driven clan tries to distribute the gathering evenly across the units, meaning there will be the same units gathering wood as the mining units.

The *GiveOrdersBuilders()* manages the computer-driven "conquering" of more components, e.g., it manages the building on different components that a player has not visited yet. The conquering splits into two stages: finding the unvisited components and finding the path to the construction.

In the first stage, the yet unvisited components for each player get found by the *GetPlayerBuildingComponents()* which returns *bool[number of players, number of map components]* having true if a player has a building on the corresponding component position. The second stage counts which units stand on a component that can reach yet an unvisited component indicated by the MapManager.ComponentsAdjacencyMatrix.

If the ComponentsAdjacencyMatrix returns true, the unit gets scheduled with construction building on that component. *GiveOrdersBuilders()* launches if a player has enough resources for building construction randomly for all computer-driven units.

The GiveOrdersDefendNearby() is a method for computer-driven defending allied buildings and units under attack. When a unit gets attacked, its attacker sends a signal to all nearby units, and when a unit is on the same team as the attacked unit, it will defend it.

6. Conclusion

This thesis shows a bottom-up RTS game's development process using the Unity engine with its DOTS technology and standard libraries. We have presented issues tied to the RTS genre, the development limitations of Unity, the selected DOTS technology, and proposed alternatives.

The RTS genre is interesting because it usually allows players to control multiple units simultaneously, where each might have specific traits or tasks to resolve, revealing challenging algorithmic/design problems like data redundancy prevention and multithreading.

The new DOTS technology raised challenges and exciting solutions for separating the game logic and data, requiring a different approach to the design than the classical Unity's GameObject/ MonoBehaviour script design. Concerning the technology selection, we have also evaluated the pros and cons of using blittable types, took advantage of the multithreaded speed using Jobs with Burst integrated within the ECS, offered solutions using alternative technologies, and deconstructed problems around data redundancy and managing multiple game units with similar data.

The crucial part of the development process was to design well-represented game environments (UI design and the game map) and optimal integration of Components and Systems under DOTS. The design later helped us to effectively implement typical RTS mechanics such as pathfinding, attacking, building, or gathering, but also fundamental logic/decision-making for computer-controlled units.

The future game extensions could offer more unit/building/resource types, more intelligent AI for controlling the enemy, the fog of war, or even changing the daytime and weather.

The ECS presents exciting solutions and designs offering different data-oriented perspectives on creating games in Unity. The game, however, can be further extended and contains only a simple implementation of RTS mechanics and a handful of unit types. The goal was not to create a new top-selling game but to walk through the game development process with exciting challenges presented by the genre and the technology selected, which this thesis summarizes.

Bibliography

[1] Assets. Unity - Manual: Asset Workflow, January 2023. URL https://docs.unity3d.com/Manual/AssetWorkflow.html. (Accessed: 16 March 2023).

[2] Baking. Convert data with Baking, 2023. URL https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/baking.html. (Accessed: 13 April 2023).

[3] Blittable and Non-Blittable Types. Blittable and Non-Blittable Types, 2017. URL https://learn.microsoft.com/en-us/dotnet/framework/interop/blittable-and-non-blittable-types. (Accessed: 6 July 2023).

[4] Burst. Burst, 2023. URL https://docs.unity3d.com/Packages/com.unity.burst@1.8/manual/index.html. (Accessed: 23 March 2023).

[5] Camera. Unity - Scripting API: Camera, 2023. URL https://docs.unity3d.com/2020.3/Documentation/ScriptReference/Camera.html. (Accessed: 4 May 2023).

[6] Cinemachine. Cinemachine Documentation, 2019. URL https://docs.unity3d.com/Packages/com.unity.cinemachine@2.3/manual/index.html. (Accessed: 5 May 2023).

[7] Common Language Runtime (CLR). Common Language Runtime (CLR) overview, 2022. URL https://learn.microsoft.com/en-us/dotnet/standard/clr. (Accessed: 6 July 2023).

[8] Components. Unity - Manual: Use components, 2023. URL https://docs.unity3d.com/Manual/UsingComponents.html. (Accessed: 21 March 2023).

[9] Components - Chunk. Chunk Components, August 2022. URL https://docs.unity3d.com/Packages/com.unity.entities@0.51/manual/ecs_chunk_component.html. (Accessed: 6 May 2023).

[10] Components - Dynamic Buffer. Dynamic Buffer Components, August 2022. URL https://docs.unity3d.com/Packages/com.unity.entities@0.51/manual/dynamic_buffers.html. (Accessed: 6 May 2023).

[11] Components - Management. Component Management, 2021. URL https://docs.unity3d.com/2020.3/Documentation/Manual/performance-memory-overview.html. (Accessed: 18 May 2023).

[12] Components - Shared. Shared Components, August 2022. URL https://docs.unity3d.com/Packages/com.unity.entities@0.51/manual/shared_component_data.html. (Accessed: 22 March 2023).

[13] Components - System State. System State Components, August 2022. URL https://docs.unity3d.com/Packages/com.unity.entities@0.51/manual/system_state_components.html. (Accessed: 23 March 2023).

[14] Components - Tag. Tag components, 2023. URL https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/components-tag.html. (Accessed: 1 May 2023).

[15] DOTS. DOTS - Unity's new multithreaded Data-Oriented Technology Stack, 2023. URL https://unity.com/dots. (Accessed: 20 March 2023).

[16] ECS. ECS for Unity, 2023. URL https://unity.com/ecs. (Accessed: 24 March 2023).

[17] ECS Components. ECS Components, August 2022. URL https://docs.unity3d.com/Packages/com.unity.entities@0.51/manual/ecs_components.html. (Accessed: 20 May 2023).

[18] Entitas. GitHub - sschmid/Entitas, March 2014. URL https://github.com/sschmid/Entitas. (Accessed: 10 April 2023).

[19] Entities. Entities, August 2022. URL https://docs.unity3d.com/Packages/com.unity.entities@0.51/manual/ecs_entities.html. (Accessed: 26 March 2023).

[20] Entities - overview. Entities overview, August 2022. URL https://docs.unity3d.com/Packages/com.unity.entities@0.51/manual/index.html. (Accessed: 25 March 2023).

[21] Entities - setup. Entities installation and setup, August 2022. URL https://docs.unity3d.com/Packages/com.unity.entities@0.51/manual/install_setup.html. (Accessed: 3 May 2023).

[22] Entities - what is new. What's new in Entities 1.0, 2023. URL https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/whats-new.html. (Accessed: 12 April 2023).

[23] EntityQuery. Querying data with EntityQuery, August 2022. URL https://docs.unity3d.com/Packages/com.unity.entities@0.51/manual/ecs_entity_query.html. (Accessed: 2 May 2023).

[24] Events. Unity - Scripting API: UnityEvent, 2023. URL https://docs.unity3d.com/ScriptReference/Events.UnityEvent.html. (Accessed: 1 May 2023).

[25] Game Objects. Unity - Manual: Game Objects, 2023. URL https://docs.unity3d.com/Manual/GameObjects.html. (Accessed: 20 March 2023).

[26] Grid. Unity - Manual: Grid, 2023. URL https://docs.unity3d.com/2020.3/Documentation/Manual/class-Grid.html. (Accessed: 15 April 2023).

[27] HDRP. High Definition Render Pipeline overview, 2023. URL https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@14.0/manual/index.html. (Accessed: 8 April 2023).

[28] Hybrid Renderer. Hybrid Renderer, August 2022. URL https://docs.unity3d.com/Packages/com.unity.rendering.hybrid@0.51/manual/index.html. (Accessed: 3 May 2023).

[29] Job system. Unity - Manual: Job system, 2023. URL https://docs.unity3d.com/Manual/JobSystem.html. (Accessed: 24 March 2023).

[30] Jobs Package. Unity Jobs Package, June 2022. URL https://docs.unity3d.com/Packages/com.unity.jobs@0.51/manual/index.html. (Accessed: 3 May 2023).

[31] JsonUtility. Unity - Scripting API: JsonUtility, 2023. URL https://docs.unity3d.com/2020.3/Documentation/ScriptReference/JsonUtility.html. (Accessed: 18 April 2023).

[32] Layers. Unity - Manual: Layers, 2023. URL https://docs.unity3d.com/Manual/Layers.html. (Accessed: 3 May 2023).

[33] LLVM. LLVM, 2010. URL https://llvm.org/. (Accessed: 20 March 2023).

[34] Marshaling. Interop Marshaling, 2023. URL https://learn.microsoft.com/en-us/dotnet/framework/interop/interop-marshalling. (Accessed: 6 July 2023).

[35] Mathematics Package. Unity Mathematics Package, March 2022. URL https://docs.unity3d.com/Packages/com.unity.mathematics@1.2/manual/index.html. (Accessed: 3 May 2023).

[36] MonoBehaviour. Unity - Scripting API: MonoBehaviour, 2022. URL https://docs.unity3d.com/ScriptReference/MonoBehaviour.html. (Accessed: 2 May 2023).

[37] Netcode. Unity Netcode for Entities, June 2022. URL https://docs.unity3d.com/Packages/com.unity.netcode@1.0/manual/index.html. (Accessed: 8 April 2023).

[38] OpenGameArt.org. OpenGameArt.org, 2009. URL https://opengameart.org/. (Accessed: 18 April 2023).

[39] Package Manager. Unity - Manual: Unity's Package Manager, 2023. URL https://docs.unity3d.com/Manual/Packages.html. (Accessed: 21 March 2023).

[40] PersistentDataPath. Unity - Scripting API: Application.persistentDataPath, 2023. URL https://docs.unity3d.com/2020.3/Documentation/ScriptReference/Application-persistentDataPath.html. (Accessed: 5 May 2023).

[41] Playback. ECS 1.0 - Entity command buffer playback, 2023. URL https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/systems-entity-command-buffer-playback.html. (Accessed: 29 May 2023).

[42] Prefabs. Unity - Manual: Prefabs, 2023. URL https://docs.unity3d.com/Manual/Prefabs.html. (Accessed: 21 March 2023).

[43] Rigidbody. Unity - Scripting API: Rigidbody, 2023. URL https://docs.unity3d.com/2020.3/Documentation/ScriptReference/Rigidbody.html. (Accessed: 18 April 2023).

[44] Margaret Rouse. Real-Time Strategy, 2015. URL https://www.techopedia.com/definition/1923/real-time-strategy-rts. (Accessed: 17 March 2023).

[45] Rule Tile. Rule Tile, November 2022. URL https://docs.unity3d.com/Packages/com.unity.2d.tilemap.extras@1.8/manual/RuleTile.html. (Accessed: 4 May 2023).

[46] Scenes. Unity - Manual: Scenes, 2023. URL https://docs.unity3d.com/Manual/CreatingScenes.html. (Accessed: 17 March 2023).

[47] Scritpting. Unity - Manual: Scripting, 2023. URL https://docs.unity3d.com/Manual/ScriptingSection.html. (Accessed: 21 March 2023).

[48] Sync points. Sync points, August 2022. URL https://docs.unity3d.com/Packages/com.unity.entities@0.51/manual/sync_points.html. (Accessed: 29 May 2023).

[49] Systems. Systems, August 2022. URL https://docs.unity3d.com/Packages/com.unity.entities@0.51/manual/ecs_systems.html. (Accessed: 26 March 2023).

[50] Systems - Creating. Systems - Creating, August 2022. URL https://docs.unity3d.com/Packages/com.unity.entities@0.51/manual/ecs_creating_systems.html. (Accessed: 11 June 2023).

[51] Systems - Update Order. System - Update Order, August 2022. URL https://docs.unity3d.com/Packages/com.unity.entities@0.51/manual/system_update_order.html. (Accessed: 11 June 2023).

[52] TextMeshPro. TextMesh Pro Documentation, December 2022. URL https://docs.unity3d.com/Packages/com.unity.textmeshpro@3.2/manual/index.html. (Accessed: 4 May 2023).

[53] Tile Palette. Unity - Manual: Creating a Tile Palette, 2023. URL https://docs.unity3d.com/2020.3/Documentation/Manual/Tilemap-Palette.html. (Accessed: 15 April 2023).

[54] Tilemap. Unity - Manual: Tilemap, 2023. URL https://docs.unity3d.com/Manual/class-Tilemap.html. (Accessed: 14 April 2023).

[55] Tilemap Renderer. Unity - Manual: Tilemap Renderer, 2023. URL https://docs.unity3d.com/2020.3/Documentation/Manual/class-TilemapRenderer.html. (Accessed: 15 April 2023).

[56] UI Comparison. Comparison of UI systems in Unity, 2023. URL https://docs.unity3d.com/Manual/UI-system-compare.html. (Accessed: 16 April 2023).

[57] Unity Collections package. Unity Collections package, August 2022. URL https://docs.unity3d.com/Packages/com.unity.collections@1.4/manual/index.html. (Accessed: 2 May 2023).

[58] Unity Components. Unity Component, April 2023. URL https://docs.unity3d.com/2020.3/Documentation/ScriptReference/Component.html. (Accessed: 14 May 2023).

List of Figures

2.1 Each Archetype has zero or more Chunks, and each Chunk hosts one or more entities of that Archetype. Chunks hosting the same Archetype will thus have the same data layout. 9
2.2 System *LumberingNavigator* fetches the positions of trees and navigates the tree cutters to the closest tree by transforming their *Translation*. System *AutoTreeCutting* then cuts the tree redistributing the wood resources from the tree to the tree cutters. . 12

3.1 The Ship would spend considerable time finding a nonexistent path to the inner *Water* circle even though the Ship and destination are the same tile type. 17
3.2 Exploring in a straight line could result in unnecessary buildings constructed to overbear components that are unnecessary to pass to get to the destination. Crosses signalize all the buildings built when a player would want to explore using this algorithm. 18
3.3 In our solution, the map falls into components. With components, a computer-driven clan can determine which components were not conquered yet (marked with red) or if the path is in the same component and exists (marked with purple). 18
3.4 Suppose a Ship (lower one) gets scheduled with a path and encounters another friendly Ship(upper one). In that case, forces between units (pink arrows) could throw the first ship off the precalculated path, and the path would need to be unnecessarily recalculated. That is the reason why forces between Ship units are so small. . . 20
3.5 *Flow-field* pathfinding 20
3.6 *Land* auto-gathering 21

4.1 *Menu* Scene shows the game settings after the user clicks on the Play button. 24
4.2 User interface after a unit gets selected. 24
4.3 A starving warning message appears if the player's food supplies are empty. 25
4.4 A ship sent to fish on a fishing spot will flow to the spot and start fishing automatically if the spot is nearby. 25
4.5 *Auto-gathering* units will automatically return to the nearest deposit building to empty their inventories if full (arrows added for clarity). The units will also automatically search for the nearest resource if their lastly selected got depleted. 26
4.6 Archers and Warriors can fight buildings or *land* units. The unit marked to attack enemies will damage them if in attack range. . . 27
4.7 Map editor 28

5.1 Structure of the **/Scripts** directory containing 6 Scripts and 4 directories. The directories are: **/Systems**, **/Components**, **/Menu** and **/MapEditor**. 29

5.2 User input from the **MapManagerMenu.cs** in *Menu* Scene initiates the start of the game, switching Scene to the *Game* Scene where the map initialization starts. The map gets initialized in **MapManager.cs**, and resources and players get spawned by the **ResourceManager.cs** and the **GameStateManager.cs**. 31
5.3 The Game Objects hierarchy in the *Game* Scene. The *Game* Scene contains: *Prefabs*, cameras, UI elements, and *MapGrid* holding 6 Tilemaps. 32
5.4 Tilemap editor on the left and the Tile palette *ground_palette* on the right. 33
5.5 *Land* Rule tile gets defined by a set of rules. Each rule is described by a 3x3 grid representing the tiles around. Different markings in the grid represent different constraints: empty space, cross, or arrow corresponds to the position/edge where the tile can, can not, or should be connected to the other neighbor tiles. 33
5.6 An example of the possible numbering of components in *Components[,]* generated by the *CreateMapComponents()* method; both implemented in the *MapManager.cs*. 34
5.7 *MainCamera* is a parent Game Object to the *MinimapCamera* in the *Game* Scene hierarchy. 35
5.8 Menu hierarchy 36
5.9 Map text file representing **W** character for *Water* tiles, **L** for *Land* tiles, **R** for *RoughTerrain* tiles, the **G** for *Grass* tiles, and **R** for *MineralDeposit* tiles. 37
5.10 *Archer* Prefab, used for Entity instancing, attached with Component Scripts like *ArcherTag*, *MovableTag* or *UnitType*. 38
5.11 *ArcherPrefab* detail, located in the *Game* Scene Prefabs hierarchy. The *Game* Scene holds all Prefabs as children under the *Prefabs* Game Object, which further separates into *Buildings*, *Units*, *Resources* parent Game Objects of individual Prefabs. These Prefabs get converted into Entities on Scene load and will be used to instantiate new buildings, units, and resources. 39
5.12 *ComponentPushOutFlowField* for pushing units out of invalid components. 41
5.13 *Flow-field* pathfinding, *Cost & Integration Fields* 42
5.14 In-game example of *Land* terrain (on the left) and the *FlowField* calculated for the *Land* pathfinding over that terrain (on the right). 42
5.15 Ship's vector field for *Water* pathfinding. This field gets created from the A* generated list of tiles representing the shortest path to the destination. If any tiles from the list have neighboring *Land* tiles, the vector forces pushing from that *Land* tile get added to the Ship's vector field. 43

List of Abbreviations

C#. C-Sharp.
RTS. Real-time strategy.
DOTS. Data-Oriented Technology Stack (Unity).
UI. User Interface.
ECS. Entity component system.
AI. Artificial intelligence.
IDE. Integrated development environment.
NPC. Non-player character.
LLVM. Low-Level Virtual Machine (initially, but today no longer stands for it).
HPC#. High-performance C#.
AOT. Ahead-of-time.
JIT. Just-in-time.
CIL. Common Intermediate Language.
CLR. Common Language Runtime.
ID. Identification.
2D. Two-dimensional.
3D. Three-dimensional.
uGUI. UnityUI.
HP. Health Points.
CC. Creative Commons.
A*. A-Star.

A. Table of controls

Control	Key
Debug mode switch on/off	X
Spawn Archer	0
Spawn Warrior	1
Spawn Ship	2
Spawn Warehouse	3
Spawn Fishery	4
Spawn Mineral	5
Spawn Tree big	6
Spawn Tree small	7
Spawn Fishing spot big	8
Spawn Fishing spot small	9
Add resources	A
Destroy selected units or buildings	R
Clear pathfinding Tilemap	C
Show pathfinding Tilemap for the selected building/moving unit	I
Show pathfinding Tilemap for pushing out of components	U
Show water auto-gather pathfinding Tilemap	O
Show land auto-gather pathfinding Tilemap	P
Camera panning left	Left arrow
Camera panning right	Right arrow
Camera panning up	Up arrow
Camera panning down	Down arrow
Camera panning speed multiplier OR If held with spawning key, the enemy gets spawned	Left Shift
Camera panning stop	Space
Camera zoom-in	S
Camera zoom-out	D
Pause menu	Esc
Action key 1	Q
Action key 2	W
Clear production queue of selected building	F
Select random idle unit	V
Select all nearby units	Left Control + Left click

B. Attached CD

This thesis includes an attached CD containing the Unity solution, the built game, and the text of the thesis itself.