BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

SIMULATION IN UNITY

SIMULACE V UNITY

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **VOJTĚCH KROPÁČEK**
AUTOR PRÁCE

SUPERVISOR **Ing. TOMÁŠ MILET**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Computer Graphics and Multimedia (DCGM) Academic year 2020/2021

Bachelor's Thesis Specification

Student: **Kropáček Vojtěch**
Programme: Information Technology
Title: **Simulation in Unity**
Category: Computer Graphics

Assignment:

1. Study Entity component system and other Unity engine techniques.
2. Design a simulation of thousands of agents with dynamic parameter changes.
3. Implement the proposed simulation and accelerate it using the studied techniques.
4. Find out the properties of the system and summarize the results.
5. Create a demonstration video

Recommended literature:

- according to supervisor instructions

Requirements for the first semester:

- Items 1 and 2 and the core of the application.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Milet Tomáš, Ing.**
Head of Department: Černocký Jan, doc. Dr. Ing.
Beginning of work: November 1, 2020
Submission deadline: May 12, 2021
Approval date: April 22, 2021

Abstract

We will build upon a recent surge in popularity of entity component systems for game development. It is our goal to create a ecosystem simulation application in the Unity engine. We will focus on the advantages of using the ECS architecture and explain its attributes and differences compared to a more traditional approach. Using the Entitas framework inside the Unity engine to power our simulation. The finished product emulates the Lotka–Volterra equations, also known as the predator–prey equations. The final findings are presented as graphs that follow the expected graphs closely. This work gives in depth knowledge about ECS architecture and its specifics in the Entitas implementation. It also summarizes my approach to creating a simple ecosystem and the findings of the simulation.

Abstrakt

V této práci budeme stavět na nedávném vzestupu popularity entity komponent systémů v oblasti vývoje her. Našim cílem je vytvořit aplikaci simulující ekosystém v programu Unity engine. Budeme se soustředit na výhody ECS architektury and vysvětlíme její atributy a rozdíly oproti tradičnímu přístupu. Pro běh naší simulace použijeme framework s názvem Entitas uvnitř Unity enginu. Dokončená implementace bude emulovat Lotka-Volterra rovnice, také známy pod názvem rovnice lovec-kořist. Nálezy budou prezentovány v grafech, které blízce sledují očekávané grafy. Tato práce poskytne hlubší porozumění ECS architektury a její specifikace v rámci implementace Entitas. Také shrne použitý přístup ke stavění jednoduchého ekosystému a výsledné nálezy.

Keywords

Simulation, Unity, Component system, Entity system, Entity Component System, ECS, Entitas, C#, Game design, Ecosystem, Composition, Composition over inheritance, Data oriented design

Klíčová slova

Simulace, Unity, Komponentní systém, Entitní systém, Entity Component System, ECS, Entitas, C#, Herní návrh, Ekosystém, Kompozice, Kompozice nad dědičností, Datově orientovaný návrh

Reference

KROPÁČEK, Vojtěch. *Simulation in Unity.* Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Milet

Rozšířený abstrakt

V roce 2020 bylo vydáno více videoher než kdy předtím. Trh s videohrami se každým rokem zvětšuje a je předpokládáno že tento trend nadále bude pokračovat. Rozpočty na výrobu špičkových her se také stále zvětšují. Interaktivní média jsou pod drobnohledem a je po nich vyžadována vyšší kvalita než v minulosti.

Programátoři videoher se tedy uchylují k novým způsobům programování. Jedním z těch způsobů je architektura, která se nazývá „Entity Component Systém“, zkráceně ECS. V této práci vysvětlím, co se pod tímto názvem skrývá. Přiblížím jednotlivé prvky, výhody i nevýhody.

Tato architektura slouží jako alternativa k objektově orientovanému programování. Základním prvkem je entita. Entita sama o sobě neobsahuje data ani chování. Ve většině implementací není entita nic jiného než identifikátor v podobě jednoduchého čísla. Toto číslo je unikátní a rostoucí. Dalším prvkem jsou komponenty. Komponenty reprezentují data, ovšem ne chování. Tyto komponenty jsou poté připojeny na entity. Tímto krokem získává entita kromě unikátního identifikátoru také jakési rozhraní, které definuje stav dané entity. Poslední součástí v názvu jsou systémy. Systémy jsou ta část, která provádí změny v datech. Systém vyčtením požadovaných komponentů specifikuje, o jaké entity má zájem, a poté na nich provede dané operace.

Většina ECS implementací zanedbává hierarchii dědičnosti tříd. Dědičnost by se neměla vyskytovat v programu jako takové a místo toho by měli entity dosahovat polymorfismu pomocí komponentů. Tento fakt má za následek také to, že ECS implementace mohou využít speciální optimalizaci, a to lokalitu dat v paměti. Eliminací volání metod skrze tabulky ukazatelů díky neexistenci virtuálních metod a následné zarovnání paměti do struktury polí místo obvyklého pole struktur můžeme dosáhnou drastického zrychlení aplikace. Při vysoké lokalitě dat se zvyšuje procento zásahu do mezipaměti na procesoru při načítání dat a tím je redukován počet přístupů do pomalejší paměti typu RAM.

Cílem této práce je využít výše zmíněný typ architektury k vytvoření simulace. Tato simulace má podobu ekosystému, ve kterém se pohybují tisíce nezávislých agentů. Tento ekosystém obsahuje tři zvířata: králík, vlk a kanec. Sdílí velkou část vlastností, počínaje nutností konzumovat potravu a pít vodu, přes stárnutí, konče možnosti se reprodukovat v rámci svého druhu.

Všechny tyto vlastnosti jsou nastavitelné v uživatelském rozhraní při prvním spuštění aplikace. Kromě vlastností specifických k jednotlivým entitám jde také nastavit parametry systémů, které působí globálně. Mezi nastavitelné parametry systémů patří rychlost růstu trávy, která slouží jako hlavní potrava pro kance a králíky, nebo například doba po které maso ležící na zemi začne ztrácet výživovou hodnotu, či dokonce velikosti iniciální populace jednotlivých druhů.

Pro udržení výkonu byly při vývoji použity různé optimalizace. V práci se soustředím na tři techniky. Jmenovitě na znovupoužití herních objektů a výhody vyhnutí se alokace paměti, rozložení náročnější výpočetní práce mezi více snímků a použití datové struktury s názvem Quad-tree, která zrychluje vyhledávání entit v prostoru.

Celá tato práce vznikla za použitím volně dostupné kostry s názvem Entitas, která poskytuje implementaci ECS paradigmatu pro Unity Engine. V textu práce je do detailu poskytnut popis této kostry a jednotlivých vlastností. Mezi nejzajímavější patří generátor zdrojového kódu, který jako reakci na velice krátkou definici třídy dokáže vytvořit detailní rozhraní pro ulehčení práce a zvýšení produktivity programátorů.

Práce také obsahuje popis mé experimentace při hledání vhodného počátečního nastavení. Při tvoření něčeho tak komplikovaného jako je ekosystém je třeba přírodu velice

abstrahovat. Tímto tedy zaniká možnost kopírovat nastavení z přírody, ale musím vytvořit vyvážený ekosystém pouze na základě vlastního pozorování.

Vyvážený ekosystém je chápán jako systém, ve kterém ani jeden z druhů zvířat nevymře chvíli po spuštění, ale ani neobsahuje zvíře, jehož populace by nekontrolovatelně rostla. Jako jeden z cílů při hledání nastavení jsem si zadal, že populace bude fluktuovat. Tedy vyšší počet králíků způsobí nedostatek potravy a jako následek tedy populace králíků opět klesne.

Povedlo se mi vytvořit systém, který obsahuje tři různé druhy zvířat a vysoké množství nastavitelných parametrů. Práci jsem otestoval spuštěním simulace po dobu dvou hodin. Výsledek této simulace byl zajímavý. Kanci dokázali po asi devadesáti minutách vytlačit králíky z ekosystému. A to i přes to že králíci začali simulaci s větší populací a mají lepší reprodukční schopnosti než kanci. Ukázalo se, že pro dlouhodobé přežití je lehce vyšší rychlost a schopnost déle přežít bez potravy důležitější.

Simulation in Unity

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Milet Tomáš. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Vojtěch Kropáček
May 17, 2021

Acknowledgements

I would like to express my gratitude to Mr. Ing. Milet Tomáš for helping me with my work, keeping me on schedule and directing me to solutions to any problems I have faced.

Contents

Chapter 1

Introduction

In the year 2020 more video-games have been released than ever before, and this number in only set to increase. Not only is the amount of video-game developers increasing, but the games released by the biggest studios look more impressive, and they are required to offer more player interactions then before.

This increased pressure led to a rapid advancement in the video game development ecosystem. One of the latest buzzwords which we will discuss in-depth in this book is Entity Component System or ECS for short. This software architecture principle has many notable attributes and is becoming more and more popular in the last few years.

But generally the benefits offered are mainly two. The first one consist of increased performance of the final application. This can manifest as increased frames per second or FPS for short. This is caused by giving the programmer the tools to optimize how application data is structured in memory. This can lead to improvements in execution speed in the order of hundreds.

The second benefit comes in the form of increased flexibility of the code. This can lead to a quicker iteration times when developing the game, or even the creation of new features without the input of programmers. The source of this benefit is focus on the *composition over inheritance* principle. This principle states that we should avoid inheritance in our programs, and instead create the final desired behaviour by combining multiple simple behaviors to create a complex entity. We will go more in depth about this principle in chapter 3.6.

In this book you can expect to gain an understanding of ECS architecture. We will start from the theoretical concepts and then later go in-depth into how this architecture can be used with the Entitas [17] ECS framework.

We will use this understanding to implement a simple ecosystem. This ecosystem should follow the predator-prey equations. The desired result is that as the number of prey type animals increases it allows the predators to find more food and thus propagate more quickly. As the number of predators in the ecosystem increases the number of available pray decreases and thus creates a cycle. This phenomenon has been observed to occur in nature.

As such an ecosystem requires a large number of agents to function properly, we will also pay attention to performance of the final application. We will go over a lot of optimization techniques that can be used to keep the simulation smooth.

Chapter 2 contains goals which I have set out to accomplish with this work. The following chapter 3 aims to provide a comprehensive overview of the Entity Component System architecture, including its positives and special characteristics but also negatives. A short overview of how a map might be procedurally generated can be found in chapter 4.

Optimization techniques that have been used can be found in chapter 5. Next chapter 6 first introduces different ECS frameworks that are available for use, following that is chapter 7 that focuses on the Entitas framework. An overview of the work done is provided in chapter 8. This leads to chapter 9 that details the process of balancing and testing the implementation. The conclusion of this work may be found in chapter 10.

Chapter 2

Goals of the simulation

My goal is to create an ecosystem with multiple different species of animals that is both sustainable and demonstrates interesting behaviours, for example similar to the Lotka-Volterra Model [21] where when you plot the amount of predators and pray the plots fluctuate.

Generally, the Lotka-Volterra Model states that as the number of prey animals increases the food available to the predators increases. This increase in availability of food leads to an increase in the number of predators. Which in turns leads to increase in hunting of pray animals thus decreasing their numbers. This creates a cycle which has been observer to occur in nature.

We have established that the final application must have at least two different animals with one acting as the predator and one as the prey. This implies that the predator must be able to hunt and kill the prey animal. As we want to visualize the application the animals need to be able to be represented with some sort of sprite and a position in space. It is also needed to add a reproduction ability to our animals so a component to distinguish between male and female specimens is appropriate.

There are a few more components and systems which are not necessarily required but make sense in the context of our application. Water is a required resource and animals need to drink it often. Animals

So, for my simulation application I have decided to extend the feature set a bit. The first major difference is that my ecosystem includes a third animal. Apart from the rabbit which represent prey and the wolf which is our predator I also included a boar which also acts as pray for the wolfs but is a bit faster than the rabbit and survives longer without food. Boars also retaliate when attacked and may even kill the predator. On the other hand they do not reproduce as quickly as rabbits.

Acting as food for rabbits is grass. While grass grows and spreads around the map quite rapidly a large number of rabbits might be able to eat most of it and thus limit their own food source. Grass may never become “extinct” as it is being planted randomly all over the map every few seconds.

While creating such a simulation of an ecosystem it might become unstable, causing one type of an entity to go extinct. For this purpose, it is important to include a large number of entities so that the findings of the simulation are more reliable and to balance the ecosystem properly so that this does not occur.

Chapter 3

Entity Component System

3.1 Architecture overview

This chapter will focus on Entity Component System Architecture. It will try to explain what this type of architecture tries to achieve and define the terms used surrounding such an architecture. ECS is a relatively new type of architecture first being defined in the year 2007 by "*Operation Flashpoint: Dragon Rising*" developer Adam Martin and his definitions are still used today. [12]

As the name might suggest this architecture has 3 main parts. These parts can be quickly summarized in the following way.

- **Entity** - Is a structure with a unique identifier that holds Components.
- **Component** - Contains data.
- **System** - Acts upon entities that have certain components and modifies their data.

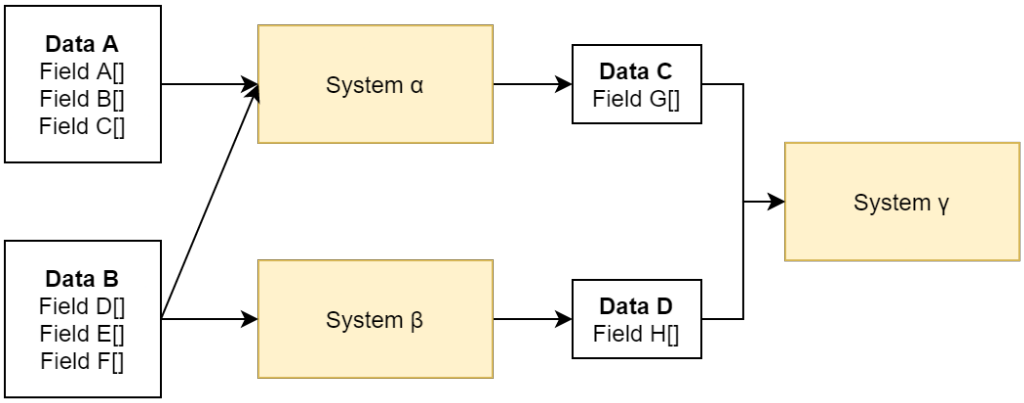

Figure 3.1: How systems interact with data. Figure inspired by [13]. This figure displays a diagram featuring three systems and four components. Systems α and β take component data **A** and **B** to create new components or modify existing data. System γ then takes the two components and acts upon them.

We will go into more specific details of every part in the following sections.

3.2 Entity

An entity is a type of object to which components may be attached. Fundamentally it is nothing more than a unique identifier (typically a 32bit integer). It does not hold any data or methods. Its only purpose is to group components together. Anything within our application may be represented by an entity ranging from a whole character, projectile, item on the ground all the way to a single frame event.

3.3 Component

A component represents data. It should never provide methods or any other type of behavior. In many ECS implementations a component is a structure or method-less class. Generally, it is a good idea to keep components as small as possible. And only group relevant data together that does not make sense to represent separately.

Typical examples include position, velocity, health, lifetime, physics or sprite components. It is up to the programmer to decide what can or should be represented as a singular component and what should be an entity with multiple components attached.

3.4 System

Systems are the lifeblood of the application. They run continuously and globally affecting any component/entity that matches their criteria. While running they may add, remove or modify components. The entities these components belong to may then exhibit different behaviors.

A simple "Hello World!"-like example is the velocity system. We have entities that hold position and velocity components. Our velocity system gathers all these entities, iterates over them and for each entity it adds the velocity vector to the position component value.

The beauty of such a system is that it does not matter if the target entity represents a projectile, a falling gold piece or a house. As long as it has a position and a velocity if should move.

It should be noted that our velocity system has no idea where, how, who or why the velocity component got added. It is of no concern for this system. It may have been added to the coin by our gravity system, by the player when he fired a projectile or when an explosion happened next to a physics enabled object.

It is also fairly easy to see how modular our theoretical scenario is already. If we want to turn off gravity, we simply don't run the gravity system. If we want to disable an object from being moved by the explosion, we add the immovable component onto it and skip that particular entity when evaluating the explosion. That we are able to alter major parts of our application without fear of breaking any dependencies is one of the main advantages of ECS design.

3.5 Goal of ECS

Overall ECS architecture tries to achieve strict separation between the data and logic. Systems do not communicate with each other and should be completely unaware of the existence of other systems. Instead, they might add temporary data to an entity. That data is later read by another system and it might affect how that system works with

this particular entity. This separation allows for maximum modularity and agility when developing software.

It can then also be argued that by removing methods from objects and instead only modifying data that the data became what is essentially the API of our application.

3.6 Comparison with Object Oriented Programming

In the modern era there are hundreds and thousands of frameworks, workflows, packages, engines and design patters. So why should you care about ECS when OOP has served you well so far?

We will get into more specific advantages and disadvantages in this chapter. But there are a few main points which can be made without going into much detail.

Firstly, most of available game engines and frameworks have used Entity Systems or EC for short for many years. This type of architecture is similar in that it favors composition over inheritance, but it places no importance on separating behavior and data. Meaning an entity is comprised of multiple components. Each component is then responsible for keeping track of its state through variables and to provide behavior or the API so that other components might communicate with it.

As games are a medium with a lot of intractability a high degree of polymorphism is needed, and EC architecture can help with that. ECS could be seen as an evolution of EC. As it keeps a similar mindset only extending and separating state from behavior even further.

Second reason that ECS architecture is desirable is that games have been known to push the limitations of computers since the early era. Game programmers that follow the ECS pattern will end up with a more optimized product or just more performance headroom which can be used to make the game more visually appealing.

3.7 Composition over Inheritance

This principle or a pattern is straight forward. An object should achieve polymorphic behavior by being composed of multiple smaller parts not by inheriting these parts.

A lot of ECS implementations forgo inheritance all together. Mainly for performance reasons as it eliminates a lot of virtual calls which can slow down the application considerably but also to be able to align memory as is explained in the upcoming chapter.

Overall inheritance should be used for very static problems and not an ever-changing environment such as the actors in a videogame.

3.8 Data-oriented design

ECS architecture lends itself well to Data oriented design. This is a type of optimization focused on memory layout and CPU cache. As modern CPUs are very fast at crunching numbers the real bottleneck in performance then becomes feeding the CPU with data fast enough so that it does not have to wait. In a typical scenario the memory an object has allocated in OOP world might be scattered all across the RAM. This leads to a lot of cache misses and in turn a whole lot of waiting the CPU has to do as it waits for data to be available to process. Performance is critical for any real-time interactable application

and by eliminating as much fetching from main memory as possible we can increase the performance of an application drastically.

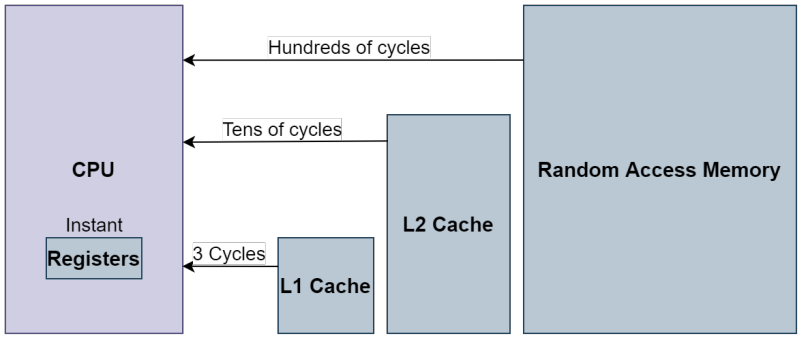

Figure 3.2: Memory timing on the PS4. Figure recreated from [9]. In this image we can see the wast difference between the speed of access to different layers of memory. Starting with registers in the CPU which provide data instantaneously and ending with RAM which can take over a few hundred CPU cycles to access.

This is achieved by having the related data close together. This is known as data locality. Any process that then loads said data from memory will load a whole cache line of data making in an order of magnitude faster to access while for example processing the next entity.

This benefit comes from storing the data relevant to our game in a structure of arrays instead of an array of structures.

The two following figures were recreated from [2].

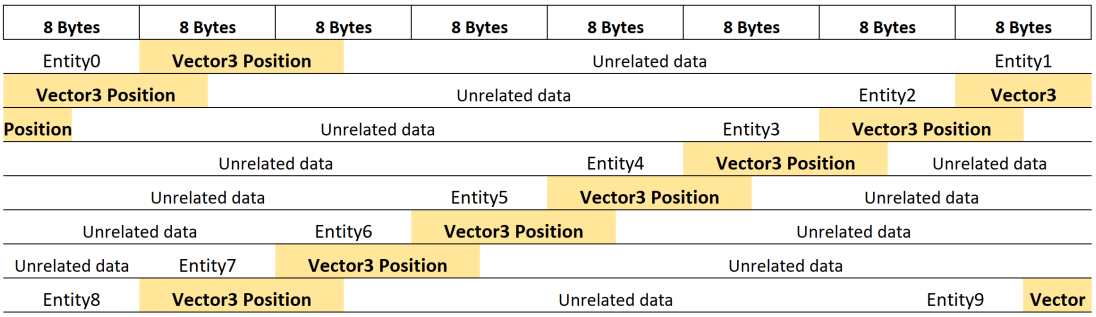

Figure 3.3: This figure represents how data might be stored in memory when using an array of structures. One line represents one CPU cache line. The data we are actually interested in is just the position of the entity and is highlighted in orange. This type of visualization clearly shows how inefficient it might be.

A memory layout like this also makes multi threading an easier endeavor. When operating on a large number of entities it is easy to split the workload and take advantage of modern processor architectures which usually provide at least 4 cores with 8 threads if not more.

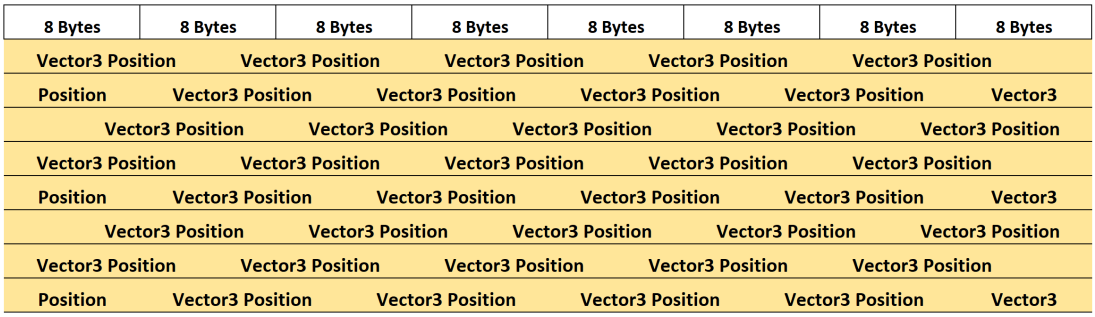

Figure 3.4: This is how data is structured when a structure of arrays is used for memory layout.

Parallelism

Also known as Multi-threading is in essence the ability to use more than one thread to calculate the state of our application. This feature became more important over time, as the number of CPU cores keeps increasing and single core speed increases have been slowing down over the past years.

There are two distinct types of parallelism [4], those are Data and Task Parallelism. Data parallelism is the ability to perform a single task upon some data spread between multiple computing cores. Opposite to that is Task parallelism which the ability to perform different tasks upon the same data on multiple computing cores.

As mentioned in the previous chapter 3.8 good memory layout can decrease the complexity of implementing multi-threading. If the memory is organized as an array of which we know the bounds beforehand we can easily split the workload and achieve data parallelism.

Similarly, if we can define which data systems are interested in reading and which data might be modified by those systems, we can achieve task parallelism by running different systems on different threads. Given that these systems cannot affect each other. In other words the systems that are run in parallel are not interested in writing into the same component or reading from a component another system might currently be writing to.

3.9 Emergent Gameplay

As defined "*instead of the behavior of the game being specifically coded in, it now emerges from a large number of variables and it's no longer always clear why certain things happen in the game.*" here [25].

ECS architecture encourages this type of gameplay implicitly as the systems a game has should be simple, have one clear purpose and operate on the smallest subset of components possible. By defining simple traits for entities to posses and then systems which create interactions between those traits may create interesting behaviours.

As an example, suppose we have an entity which represents a tree. We decide we want to be able to create forest fires, so we create a flammable component and attach it to a tree. This component contains the temperature at which point an entity caches on fire. When an entity reaches a certain temperature, we add the burning component. A flame system then iterates over every entity with the burning component and increases the temperature of nearest entities. The emergent component comes in the form of other flammable entities. Imagine we have a player. He is not flammable per say but does take damage when at a

high temperature. What than might happen is that the clothes a player wears are entities upon themselves which are flammable. This results in the player catching on fire when near one for a long time and taking damage from it. Even though there is no one specific system or component that deals with the player being on fire.

This aspect of the ECS paradigm can also be negative as it may make debugging the final product a difficult endeavor.

Chapter 4

Procedural map generation

Procedural generation in games is defined as "*an algorithmic creation of game content with limited or indirect user input*" by the following book [20]. In this project I will be using procedural generation to create the gameplay area for our agents to inhabit.

4.1 Types of noise

If an algorithm is used to generate content, the same result would always be reached. In my case that is undesirable so randomness or noise needs to be a part of the generation process.

There are many different types of noise [1]. When using noise for generation it is desirable for it to be continuous. In other words if the noise function returns smooth results or pure noise. Pure noise usually does not produce the desired results as it is too random.

There are many different noise functions. Notable examples include:

- **Perlin noise** - Produces a smooth gradient. Many other noise functions build on top of Perlin noise.
- **Cellular Noise/Worley Noise** - This algorithm generates pattern that look like an organic cell.
- **Simplex noise** - Created by the author of Perlin noise. Includes some improvements to the original.
- **Distribution noise** - Many different variation including Exponential, Binomial or Poisson noise.
- **Vorinoi noise** - A different type of noise that creates organic like cells.

Different noise functions provide better results in different scenarios. For this project I have chosen Perlin noise as the preferred method to create terrain height map.

4.2 Perlin noise

Perlin noise in known as a continuous noise function. In our case it is necessary to have a continuous noise so that the final map looks more realistic.

Fractal Perlin noise uses multiple octaves at varying amplitudes and frequencies to generate the final result. Multiple octaves allow us to emulate nature more closely. Each octave has a different frequency There are many different tweak-able parameters while using Perlin noise.

- **Scale** - Sets the detail level of the final noise map.
- **Octaves** - Number of additional higher frequency Perlin noise functions being used.
- **Persistence** - Each subsequent octave amplitude is multiplied by this number.
- **Lacunarity** - Each additional octave frequency is multiplied by this number.

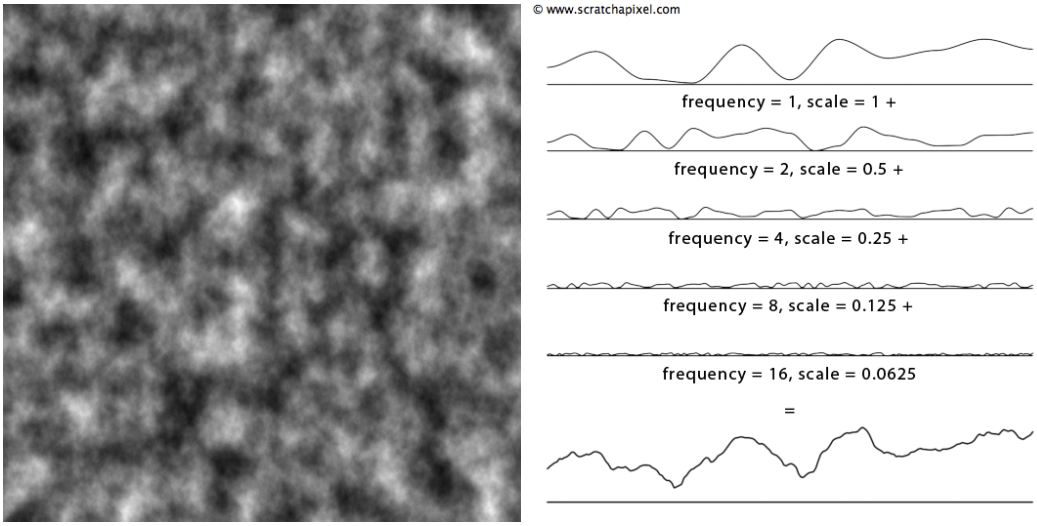

Figure 4.1: a) 2D Perlin noise mapped onto a black and white texture b) 1D fractal Perlin noise showcase taken from [14]. Layers are displayed at different frequencies and then added together for the final noise signal.

4.3 Height map

After we create a 2D perlin noise map as seen in figure 4.1 we can use it to generate a map. We do this by mapping sprites onto the map. A completely black pixel in the map signifies the lowest spot so we assign it to look like water. As pixel get more gray and white we transition into assigning sand and later to grass. Essentially using the noise as a height map.

To get rid of repeating patterns on large areas of grass, I decided to include multiple different grass sprites with some containing flowers and the like.

After the height map is generated trees are randomly placed in the environment to increase the visual appeal.

Chapter 5

Optimization

As stated in chapter 2 we are targeting thousands of entities for our simulation. With such a high number of entities the performance of our final application becomes critical to optimize.

5.1 Unity tilemap

Unity engine provides a tilemap feature [23]. This feature handles the rendering of sprites on a 2D grid. Essentially its main advantage over just assigning a separate entity to each tile is that no new game objects need to be created which improves performance.

5.2 Object pooling

This optimization is quite common among garbage collected languages and the Unity engine applications in general. The goal of this approach is to avoid allocating new memory and thus avoid the slowdown that occurs when the garbage collector runs.

This is achieved by not deleting objects. Instead of deleting and then freeing the memory that has been allocated we keep the object in memory and re-use it. This can be done in many ways but the simplest one is to mark such an object as "inactive" with and hide it.

When the need to instantiate a new object arises we can simple take and object from our pool. This pattern is especially useful when dealing with a large amount of objects being created and deleted. A very common example are projectiles. In any game featuring a shooting mechanic the projectiles may be fired multiple times per second. The strain the allocation and later deallocation of memory would put on the system could be the difference between a smooth pleasant experience and an unpleasant one which stutters.

As video-games are more sensitive when it comes to hiccups it is especially important to use this pattern when using a "Stop-the-World" type of garbage collector.

Unity engine has been using a garbage collector of such a type for many years but since 2018 it is possible to switch into a "incremental" garbage collection mode and thus avoid big spikes. You can read more about the incremental garbage collector here [6].

Thankfully, object pooling is a key feature of the Entitas framework which I used when developing this project and thus a lot of performance is saved without any work already. Although entity visuals such as moving sprites still need to be pooled manually.

If you are interested in reading more about memory management in Unity you can do so here [22].

5.3 Time slicing

This technique is quite simple to understand. When doing calculations on the CPU they should be distributed among multiple frames so that no single frame takes significantly more time to calculate [3]. I have achieved this by randomly offsetting all the entity updates within my application. You can see the result in figure 5.1.

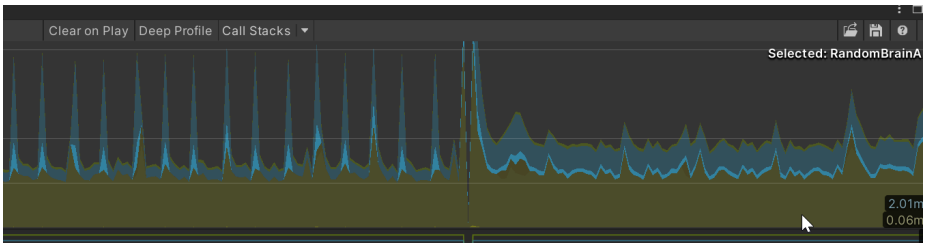

Figure 5.1: Screenshot taken from the Unity Profiler which is a tool in the Unity Engine that servers to debug performance issues. On the Y axis is time taken to calculate a frame while the X axis shows the history of previous frames. This figure clearly demonstrates the advantages of splitting the work among multiple frames. The left side displays multiple spikes that interrupt the user and cause an unpleasant experience every few frames. The right side of the graph displays the work being evenly distributed over time with no noticeable spikes in performance.

5.4 Quad trees

It is often the case that during runtime we would like to query if an entity is present within an area. A naive approach would be to iterate over all relevant subjects and check their positions. But this approach would quickly bottleneck our application in terms of speed.

We can speedup this search by implementing a Quadtree data structure. As the name implies it is a tree-based structure meaning it is created by recursively parenting nodes which together creates a list of references from the root to the leaf node. While there are many different implementations of the Quadtree data structure they all share the same idea and most of the attributes.

In a Quadtree every node has four or zero children. Each node keeps a list of entities it tracks. When this list grows too big the node splits. It spawns four children. These children nodes all have authority over one quarter of the space of its parent. When a query is then received to retrieve an entity based on its position all a node needs to do is check if any entity from its internal list matches the criteria and then relay that query to child nodes and append the results.

Because a node knows the bounds of its authority it can very easily know if it is even possible that there exists an entity that matches the query criteria. This increases the search speed considerably as we can skip evaluating the condition on entities that cannot fit the condition.

Quadtrees have been used in image compression, Game of Life simulations, View frustum culling or even spreadsheets. There are variations called Octrees that deal with 3D space or k-d trees that deal with k-dimensional space.

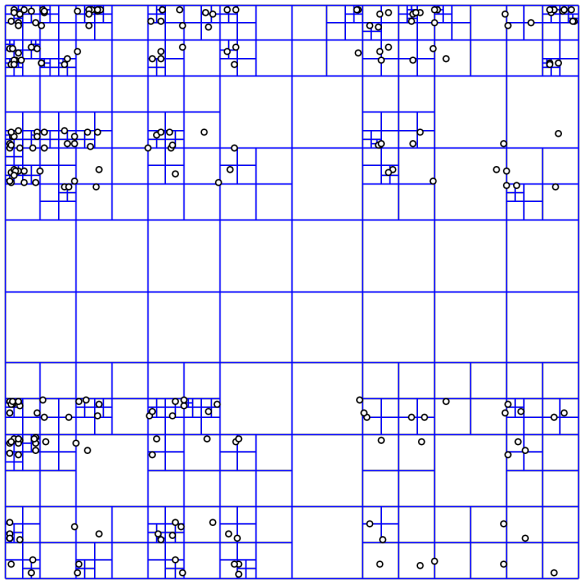

Figure 5.2: Quad tree visualization. Clearly showcasing how points on a 2D plane are grouped into nodes. These nodes get smaller and smaller as more points are added.

In my application I have decided to go for a simpler implementation with only one layer. This type of implementation is simpler but sufficient for the amount of entities that need to be tracked.

Chapter 6

Frameworks

This short chapter serves as an overview of the currently available frameworks that I have considered to use for the development purposes of this project. It is not in any way a complete overview and does not provide all the specifics of frameworks mentioned. Its purpose is to give perspective on the current landscape of all different possible ECS implementations that are available to the public as of writing of this thesis.

Note that all the frameworks featured are compatible with the Unity engine. In chapter 7 we will go in-depth about the framework I have chosen to implement this project in.

6.1 Unity Data Oriented Tech Stack

Unity Data Oriented Tech Stack[24] or DOTS for short has been revealed by Unity Technologies in 2018. The term "tech stack" is used to indicate that multiple different technologies are included. Among these technologies is an ECS system, but also a C# Job System, Burst compiler and others. The goal of this tech stack is to allow developers to create massive games without having to worry about performance.

Over the last few years Unity Technologies has been improving this tech stack and collecting feedback from the public. It is not yet recommended for production.

When using this technology, you might stumble upon crashes, missing API documentation or outright broken features. This has been confirmed by my experience where a simple scene with a few sprites and a system that moves them around would perform erratically and not work at all once the project was build.

However, if your application requires the best performance available and you want to use the Unity engine then this framework is the right choice. Multiple games have already been released using D.O.T.S as the main driving force or just in specific parts of its architecture.

6.2 SveltoECS

Svelto framework [18] is a C# that uses the Entity Component System architecture. It has been used to ship three feature compete games by the company where the creator of Svelto ECS works.

The framework receives updates quite often with new features added regularly. The author of this framework also publishes lengthy articles which go into the thought process behind Svelto and its features on his blog [19].

As demonstrated by the blog its creator cares about performance and this fact is demonstrated with how optimized the framework is. In some areas even exceeding the Unity Technologies implementation. It also offers deep profiling and debugging support.

It should be noted however that the community around this is quite small. This shows especially when searching for examples or documentation. The author is active on his Discord server and its possible to get firsthand support from him.

6.3 Other frameworks

What follows is a list of less popular frameworks upon which I have stumbled with their main features pointed out. They have been sorted by the last update date at the time of writing this book.

- **LeoECS** [11] - No dependency on Unity Engine, automatic data injection, structure based.
- **Morpeh** [15] - Focus on mobile casual games, no code generation, single threaded, structure based, uses C# generic types, MIT license.
- **NanoECS** [10] - Visual debugging, code generation, reactive components.
- **Actors** [5] - No code generation, pooling, multi-scene editing, reactive variables, first class integration with Unity engine, uses C# generic types, code generation, reactive components, parenting entities.
- **EcsRx** [7] - Reactive approach to ECS, Engine independent, dependency injection, testable and mock-able, visual debugging.

Chapter 7

Entitas

Entitas [17] is a class-based Entity Component System Framework specifically made for C# and Unity.

This section's structure and content was inspired by the Entitas Cookbook [27] which is a short guide or summary created for those interested in learning this particular framework.

It has several implementation specifics. Some of them are focused on speed of the final application and some are oriented to help create readable code quickly.

Among application speed optimizations in a caching system that increases the speed at which you can access internal components and things like pooling to avoid the performance hit of a garbage collected language.

A part of the framework is also a code generator that is one of the main features of Entitas. It allows the developer to access a very comprehensive API for the purpose of development without having to write a lot of code, instead the developer can rely on code generation to create it.

7.1 Code generation

This framework is distributed with a custom source code generator named Jenny. Source code generation can be used to decrease the amount of code a programmer has to write while working with this framework. After successful compilation where a file was added or modifier the generator can be run by the user. If it's appropriate the generator creates or edits exiting previous source files that have been generated to fit the new definition.

Entitas uses code generation to provide an API for each and every component or context that has been specified. It respects a lot of attributes that can be added to a field of a class.

```
[Game]
public class MovableComponent : IComponent {}
```

This small bit of code would create methods to assign this component to any field in the [Game] context. We will discuss contexts in the next section.

```
GameEntity e;
var movable = e.isMovable;
e.isMovable = true;
```

Simple components that do not contain fields are called flag components. They provide information about the entity just by mere presence and amount to a Boolean value. More complex components might contain fields inside and have a bit more involved API.

```
[Game]
public class PositionComponent : IComponent {
    public int x;
    public int y;
    public int z;
}
```

A more complicated class like this Would generate an API with methods to add, remove or replace the component. And a method to check for the presence of said component.

```
GameEntity e;
if(e.hasPosition)
    var position = e.position;
else
    e.AddPosition(x, y, z);
e.ReplacePosition(x, y, z);
e.RemovePosition();
```

The code generator also creates a class with a list of component names, types and indices. The indices are used when accessing the components. Specifically, they are indexes into an array of components every entity has. This array contains references to the memory where component data is stored. By using an index and an array we can avoid garbage allocation and increase the performance of our application rather than using a dynamic data structure like a *List*.

I would also like to note that source code generation can cause some issues or friction. The main drawback is that the generator cannot be run if the project is not able to be compiled. This can also hinder refactoring of components. Even something as simple as a renaming of a field has to be done manually.

7.2 Contexts

A context is a concept that is shared by many ECS implementations. Other ECS implementations have called this Universe, World, Scene, Layer or Entity group.

Context in Entitas represents two things.

First, it is a type of manager that provides access to all entities contained within that context, it serves to create new entities (with *context.CreateEntity()*), delete them when appropriate and pool them for later use. Pooling is described in chapter 5.2. A context also guarantees that if proper methods are utilized the user can work with references to entities that will prevent the re-use of said entity until no other entity holds a reference to it. This is done with simple reference counting and two methods.

```
entity.Retain(this);
entity.Release(this);
```

A comparison can be made to relational database when using Entitas. A component represents a column in such a database, an entity is a row, and the context is the whole table. In such a comparison the classes that implement the “IComponent” interface together make up the schema that defines the database. As we increase the number of components

the table column count increases. As each entity is backed by a pre-allocated array this increases the memory required to store the data of each and every entity.

This leads us to the second feature of contexts. An entity within said context can only have components that share the same context definition. This allows the system to decrease memory consumption as the backing array with references to each component can be smaller.

When two components can never be part of the same entity they should probably be in different contexts.

An example of this is the difference between [UI] context and [Game] context. While [UI] can be clicked and receive input same as the entity in [Game] context. It is probably safe to assume that [UI] entities will never be attacked and thus do not need to have a health component or armor component.

While it is not necessary to categorize components into different contexts it does make the code more readable and decreases the memory requirements and therefore performance of the final application.

7.3 Message entities

When developing applications with ECS paradigm its sometimes needed to exchange information between entities or apply one-time events.

For this we can use message entities. Message entity is a type of entity with a very short lifetime (usually only one frame) that represents an event. We create this entity in what is preferably a context specially designated for such entities.

A very simple example is damaging an entity. We create an entity that holds a reference to the target, the amount of damage to dead and maybe other meta data like the source or type of damage. When such an entity is created, we apply it in a system and delete it.

This way of handling communication between entities is very flexible and at the same time offers a lot of control. We can have multiple systems that react to different events or even the same event and don't delete the entity. It should be noted that even with pooling this way of communication can be less performant that other ways.

7.4 Groups and collectors

As we have discussed previously in the ECS overview chapter. Systems are scripts that act upon a subset of entities based on the presence, absence or state of its components. To decrease the amount of code required to specify on which entities to act Entitas has Groups.

```
IGroup<GameEntity> targetGroup;
targetGroup = context.GetGroup(GameMatcher.AllOf(GameMatcher.Position,
GameMatcher.Velocity));
```

This piece of code creates a Group on the specified context and assigns it to a variable. That group will contain all the entities that have both a Position and Velocity components.

A Group always stays up to date. It is therefore important that if we are removing either of these components while executing such a system, that we use the "GetEntities()" method otherwise we might encounter a "Collection was modified" at runtime.

Matcher

You will notice that the previous example included a few references to "GameMatcher". It is a class generated for every component in every context that's used to specify which components we are interested in. As is apparent from the name the "GameMatcher" is generated for components contained within "Game" context.

There are three specifiers we can use when using matchers.

```
Matcher.AllOf();
Matcher.AnyOf();
Matcher.NoneOf();
```

And we can chain such matchers together with the only exception being "NoneOf" which cannot be the first matcher in a chain.

```
context.GetGroup(Matcher.AllOf(Matcher.A, Matcher.B)
                        .AnyOf(Matcher.C, Matcher.D)
                        .NoneOf(Matcher.E))
```

Collectors

Collector is a class which we can use to observe and listen for specific changes within a group. Essentially limiting when our system runs to only specific events. These events include "Added", "Removed" and "AddedOrRemoved". We can use a collector in the following way.

```
context.CreateCollector(GameMatcher.Position.Added());
```

It is a shorthand way to subscribe to an "Added" event which the group emits whenever an entity is added to the group. It is possible to specify multiple events or groups essentially creating a joined list.

7.5 Entity indexes

When creating an entity, it can become quite difficult to keep track of specific entities or to query for a subset of them without iterating over every single entity. This is where indexes come in.

There are two types of indexes which provide the developer with an ability to keep track of or reduce the amount of iteration when searching for a specific entity.

We can create an Entity index by marking a field within a component with an attribute.

We have "[PrimaryEntityIndex]" which guarantees the uniqueness of said field withing the whole application and "[EntityIndex]" which just allows to query for an entity by the value inside the marked field.

```
[Game]
public class IdComponent : IComponent {
    [PrimaryEntityIndex]
    public int value;
}
```

```
[Game]
public class InventoryItemComponent : IComponent {
    [EntityIndex]
    public int ownerId;
}
```

Guaranteed uniqueness is quite helpful especially because Entitas pools entities and therefore references might become invalid if the reference counting system is not properly utilized.

By ensuring that the ID component is unique by counting from zero and incrementing every time an entity is created or removed from the pool of entities, we don't have to rely on the reference counting system and instead just query for the existence of an entity with the specified ID value.

In the background every entity has a reference kept in a dictionary structure where the index (ID) is the key, and the value is the entity. This allows for searching in $\mathcal{O}(n)$ linear time. Meaning the time to find an entity is independent of the number of entities within our application.

Just using plain [EntityIndex] does not guarantee uniqueness. But it can still reduce lookup time for entities with specified values. As is shown in the example. We might have a component which denotes an item as being in the inventory of a specific entity. It is then quite simple to get all the entities that we own and displaying them in UI. Using an entity index instead of iterating over all entities within existence we can dramatically reduce the computation time.

In the background an entity index is implemented as a dictionary where the key is the index (position) and the value is a hash set of entities. This way we are essentially saying there might be more than one result when querying a position. The speed is lower and memory requirements are higher than when using [PrimaryEntityIndex]

```
context.game.GetEntityWithId(idToSearch);
context.game.GetEntitiesWithInventoryItem(idOfOwner);
```

Chapter 8

Simulation implementation

In this chapter I will present how the final application is put together. Spotlighting and explaining key systems that represent most of the functionality of the game.

8.1 Initialization

When the application is first launched, it needs to be initialized. Among the first systems that are run are *UnityMapService* and *UnityViewService*. As the names suggest these are services used to integrate entities from our simulation with the Unity engine. We use services to separate our simulation from the display logic. Essentially making the simulation completely unaware how or even if its being displayed.

After our services are initialized, we generate our map in the *GridInitializeSystem*. The process for generating our map has been overviewed in the map generation chapter 4. Among other things this system also launches a method that prepare the positions from which entities may drink water.

Lastly the *EntitySpawner* system is run. This system places our entities within the world. It also attaches all the appropriate components to them.

8.2 Update Systems

The simulation features two different update systems. The first system runs every frame and serves to update values that represent continuous or analogue features like time, hunger or thirst.

The second one is something I called a *BrainUpdate*, only entities which have a *Brain* component may receive this update. This type of update happens only once every second or so and represent a major change to the state of an entity. The entity scans its surroundings, decides which type of action to take next and changes its position in accordance with its intentions.

8.3 Artificial intelligence

Artificial intelligence or AI for short in video games [16] is not what experts in traditional AI field are researching. Games have wildly different requirements for the feature-set compared to AI as viewed by the public and experts.

Traditional AI does not cheat and instead relies only on its own senses whereas it might be completely acceptable for game AI to receive information about the players exact position or something similar.

It is not important for the game AI to solve every situation or be able to act realistically. It is instead encouraged to act in a way that makes the game fun for the player.

Game AI needs to be able to evaluate the situation quickly and without wasting resources especially in real time games like shooters or real time strategies. For this many simplifications and abstractions are needed.

The topic of AI is quite unexplored in the domain of ECS game architecture. Traditionally there are multiple approaches to handling AI agents [8] but no specific pattern or architecture has yet been defined that works especially well with ECS.

Finite State Machines

One of the tried and tested methods is Finite State Machines or FSM for short. This approach can be represented as a graph containing multiple states and oriented transitions between them. Each agent then has a state which represents its current behavior. This state has transitions going in or out of it to other states. We check the conditions assigned to each of these transitions every time and either keep the current state or change it appropriately.

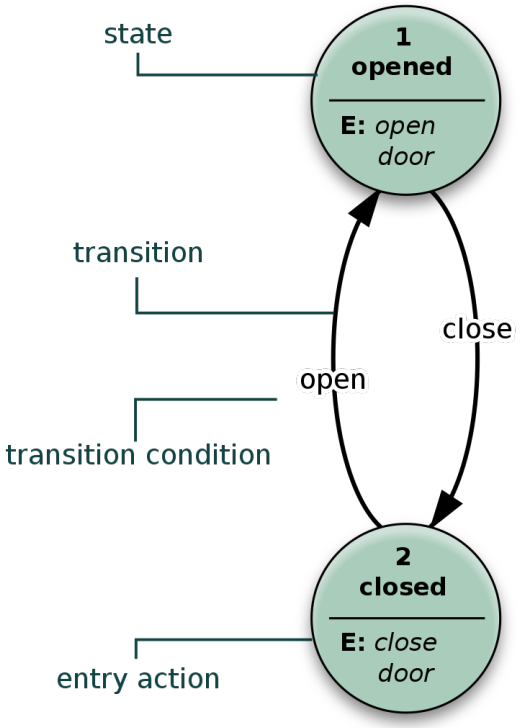

Figure 8.1: A visualization of a simple FSM. Consisting of just two states and two transitions. Representing a door state machine.

Behavior trees

Behavior trees are tree like structures which are evaluated from the root node going left to right within the children of every node.

Overall, there are four types of nodes. The most basic one is an *action* node. This type of node is always at the leaf level of the tree and serve as basic building blocks.

Composite nodes are those that have more than one child. We have two distinct types. First one is a *sequence* node which executes its children but if one child fails it also fails. This means that all children must succeed in order for this node to be successful. The other type is a *selector* node which acts as a sort of opposite to the *sequence* node in that it executes its children and succeeds as soon as one of its children succeeds.

Last important type of node is the *decorator*. This node may only have one child and changes the behavior of such a child node. An example of a decorator node is a *repeater* node which runs its child node a number of times or an *inverter* node which inverts its child success result.

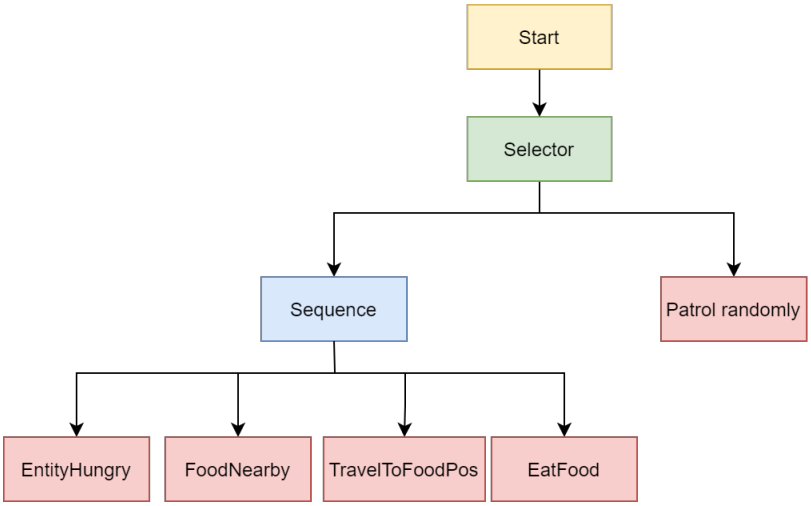

Figure 8.2: Reading from the start node. We can see that we will either have to succeed at the whole sequence to the left or fallback to the *Patrol randomly* action. The sequence consists of checking if the entity is hungry and if so, we continue to check if there is food nearby. After both checks succeed, we move to the food and eat it.

Utility AI

Also known as Goal Oriented Action Planning or GOAP for short is the third and final major AI type that I have considered implementing for this project. It works be scoring all the actions available to the agent based on the desirability of the outcome of such an action.

But ultimately this type of AI turned out not appropriate for a very large number of agents as the performance would suffer. Because a single agent then may consider multiple different solutions which mean with thousands of agents updating every single second the final number of queries might reach tens of thousands. Without significant focus on optimization this approach would not be sustainable.

Simple conditional statements

All the previously mentioned approaches to handling AI are valid and can represent anything from a very simple to incredibly complex behaviors.

But after considering all of the possible approaches described above, I have decided to not use any of them. I instead chose a much simpler to implement conditional branching AI.

The biggest reason was that there are no libraries or frameworks available for use with the ECS architecture in mind or Entitas. This means that I would have to implement a whole AI library which is out of scope for this work.

This approach of handling AI through a simple series of branching if conditions is not as flexible or extensible. But for the purposes of my simulation it provides the best possible optimization and all the behaviour in this work are simple enough so as not to get unwieldy.

The main idea behind my implementation is that there exist a single system called *AIBrainBehaviourChangeSystem* that handles all of the AI decisions. This system first considers if an animal is hungry or thirsty, if so it queries its environment to see if it can satisfy that need by either eating grass or meat or even start hunting another animal. If at any point the AI cannot satisfy a need it falls back to wandering around the map randomly. If the entity is male and both its hunger and thirst needs are met it might consider searching for a mate to reproduce with.

After an appropriate behaviour is chosen the system assigns a special behaviour component. While an entity has a behaviour component it does not consider changing it until the goal of that behaviour is met or it fails.

Acting upon entities with these behaviours are specific systems for each one. These systems are responsible for detecting whether the goal has been reached or is invalid. For example a system called *UpdateWaterSeachBrain* moves an animal towards the nearest water source and when that destination is reached it removes the *WaterSearchBehaviour* and instead applies *DrinkWaterAction* component to that entity. Action components are described in the next section 8.4.

8.4 Action systems

An entity might perform an action that is not suitable to be abstracted as being instant and should instead take multiple frames to resolve. For this there are systems that handle these actions.When we add an action component to an entity, we mark that entity as *IsPerformingAction* and skip updating its brain.

For example, when an animal reaches a food source it starts decreasing how hungry it is and decreasing the amount of food left in the food source. This process takes a variable amount of time that is based upon how fast the animal gets full or if the food source runs out of food to provide. This scenario is handled by action systems which update the relevant values and when appropriate remove the action component from the animal and the inhibition on updating its brain.

8.5 Cleanup systems

At the end of the update loop, we run systems that perform cleaning up. This mainly means removing any temporary components that were used to transfer data between systems and message entities.

8.6 Extension options

Due to the large scope of this project not every feature was implemented. While working on this project many different extensions and features had been considered. The following list represents rough ideas which might be valuable to the simulation and the usability of the application if it's going to be further developed.

- **Predator fear** - A feature that would make prey run away and hide from predators or at least try to keep a safe distance when wandering around the map.
- **Genetics** - When a child is born mutate one of its parameters randomly within a certain range. This would serve as a simple evolution mechanic.
- **Advanced tracking** - Much more data can be extracted from the simulation. Starting from heat maps and all the different entity parameters when they perform certain actions.
- **Initial entity positioning** - Its unnatural to place all the plants and entities randomly. For example, rabbits and wolfs should spawn in packs, trees and other plants should also be roughly grouped by species.
- **Limit bush growth** - Currently grass growth is not limited. It might be interesting to implement some sort of system that calculates the nutrition that is available in the ground around a plant. This would prevent excessive growth and might also give rise to other emergent behaviour.
- **Better ways to acquire water** - Currently water can only be refilled when at the shore of a lake. As these lakes are limitless this mechanic does not provide much in the way of interesting mechanics or behaviour for the entities.
- **Day and night cycle** - A system that would handle the passage of time. Represented as a change in the lighting conditions which might affect the sensory range of some animals and their overall behaviour. Further expanding on this would be a system that changes the season.
- **Home or nest system for animals** - Animal behaviour is currently very random with no permanent point of residence.
- **Gamification** - The application in general might benefit more from higher degree of user interaction rather than just setting the initial parameters and watching the simulation play out. [26]

Chapter 9

Findings

This chapter will overview my process of tweaking the initial simulation settings, I will also give examples of interesting consequences of changing different values and explain why the results are what they are.

An ecosystem has two very distinct states. Either it is unstable or stable. A stable ecosystem stays within reasonable values and expectations. Opposite to that is an unstable ecosystem that might for example cause total extinction of a species or an unreasonable growth of a certain population seemingly without limit.

An analogue to the action of trying to find stable settings for a simulation with multiple different entities, interactions and systems might be found in mathematics, that is with trying to find a solution to an equation with multiple variables. This search for some sort of balance in the system cannot be done automatically and the only real help when doing this is common sense and insight into the systems at play.

9.1 Unstable populations

When balancing an ecosystem it is important to get the most basic elements tuned well first. In my simulation that is the population of rabbits. This population is limited by the availability of food, water, their reproduction speed and age limitations on reproduction.

A graph that displays an early extinction of rabbits can be seen in figure 9.1.

9.2 Fluctuating population

Creating a stable population with just rabbits and grass is not very interesting. My next goal after getting a sustained population was to introduce fluctuation. I wanted to create a system where as more rabbits are born the rate at which grass is eaten exceeds the rate at which new grass is grown. This creates a shortage of food and thus rabbits start dying of hunger.

Figure 9.2 gives an example of a simulation that is quite stable as the rate of deaths does not change over time. This signifies that the rate at which rabbits are birthed is very similar to how often they are dying. After some tweaking and experimentation with the settings of the simulation I was able to create a fluctuating population as shown in figure 9.3.

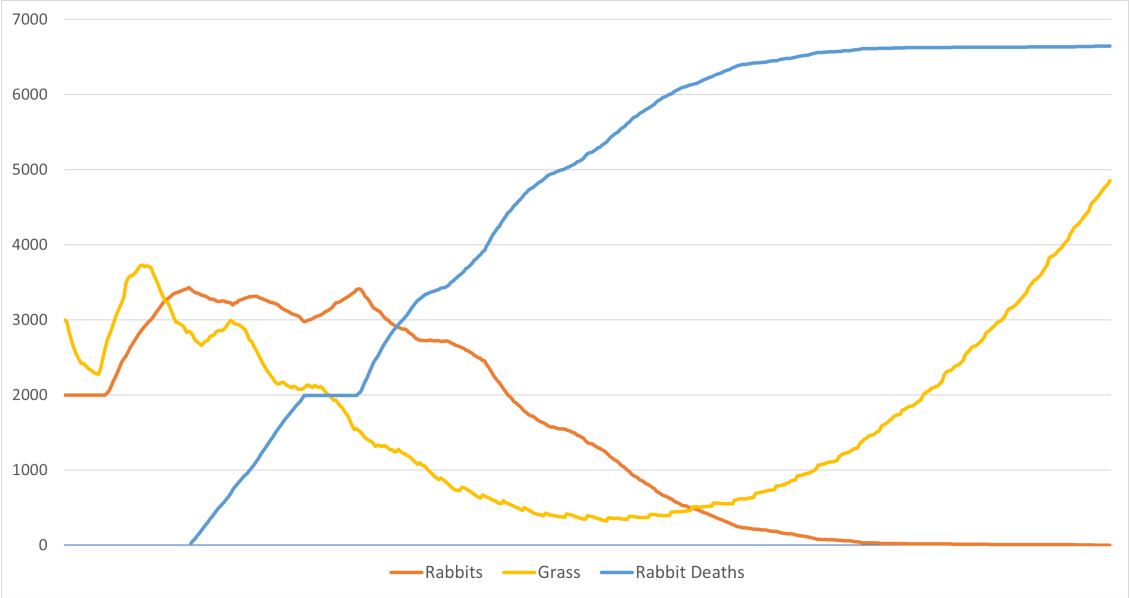

Figure 9.1: Figure displays an unstable rabbit population that goes extinct after just one generation. This was caused by a long delay between gaining the ability to mate with other rabbits, short lifespan and small amount of new rabbits being born.

9.3 Predators liability to go extinct

Because predators depend on prey for their sustenance they are much more susceptible to a lack of food. Because the amount of food available fluctuates heavily especially in the first moments of the simulation it is possible that the predator species goes extinct right in the beginning. As seen in figure 9.4 this decrease in food leads to the wolfs not reproducing. When the rabbit population starts recovering it is already too late because all of the wolfs are too spread out around the map and thus its impossible for them to find a suitable mate.

9.4 Uncapped population growth

Even a small change in the simulation settings might cause unforeseen consequences. During my experimentation I have encountered a good example of this.

While trying to solve the problem that I indicated in the previous section I decided to increase the food amount a wolf drops when killed. This small change resulted in uncontrollable population growth.

The reason for this growth is that when a wolf died. The amount of food it added was enough to sustain a newborn wolf all the way to adult hood and for it to reproduce. This essentially created a positive feedback loop, thus generating resources inside the system with seemingly no limit. Figure 9.5 show how that growth might look.

9.5 Survival of the fittest

I am using this term loosely here. As a final experiment to truly test how balanced or imbalanced my ecosystem is I ran a two-hour long simulation. The results of which you can see in figure 9.6.

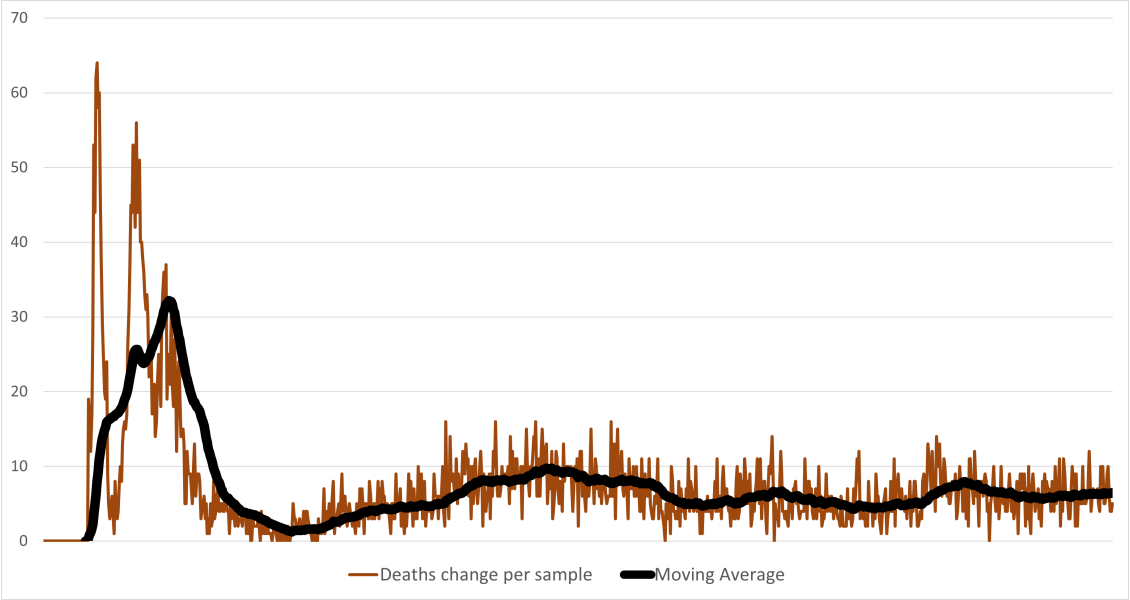

Figure 9.2: A graph displaying the amount of deaths between measurements and a moving average. It shows very little fluctuation in the amount of deaths and is generally steady. The peeks in the first few samples are caused by rabbits that were spawned in the initial step of the simulation and are dying of old age.

Overall, I would call this experiment a success. After about 45 minutes wolfs went extinct. As their reproduction speed, is slow they never managed to increase in numbers only keeping a stable population. It took over 90 minutes, but rabbits went extinct. As there are no predators in the system the only explanation for rabbits going extinct is that they have been overtaken in the food-chain by boars. Boars have the same diet of grass but are just a bit faster and can also last longer without food. Even though they do not reproduce as quickly as rabbits after a significant amount of time they managed to push the rabbits out of the ecosystem.

9.6 Simulation length

As the ecosystem becomes more and more balanced it takes more time to see the results of small tweaks. And it becomes even harder to find all the side effects introduced. Because of these reasons I have decided to stop testing and balancing the system after running a two hour long experiment as described in the previous section.

The problem of waiting for results to be gathered can be rectified by increasing the simulation speed. But this requires more computing power.

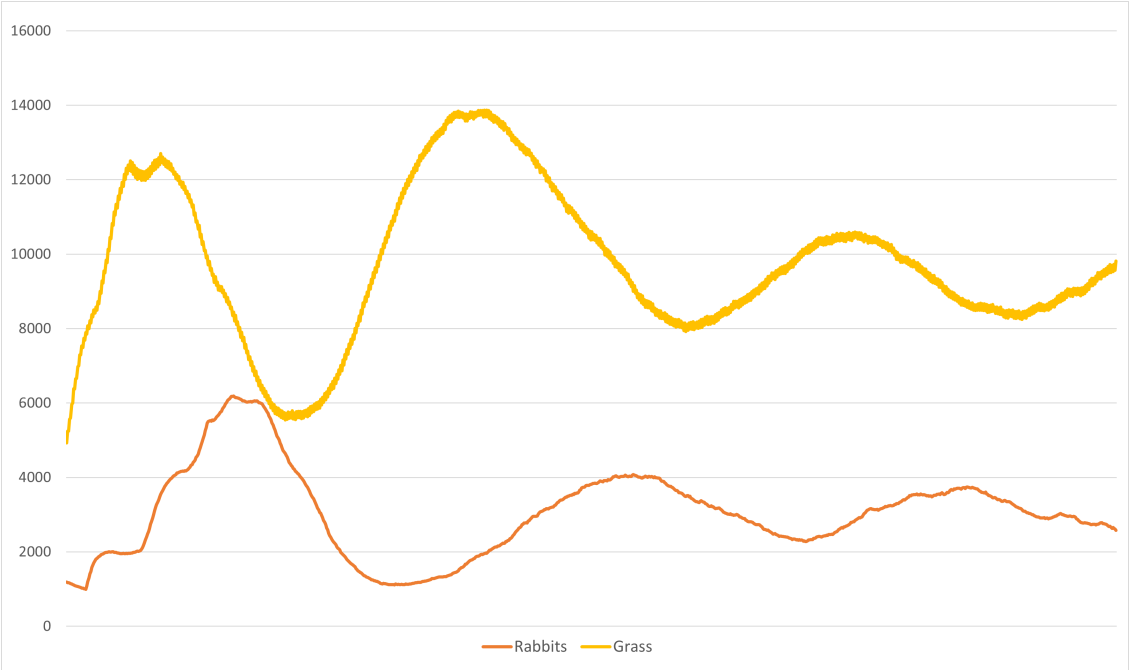

Figure 9.3: Decreasing the maximum food capacity a rabbit may hold and decreasing the amount of food grass provides creates fluctuations in the population. Over time waves like this slowly even out. This happens because the map contains multiple groups of rabbits and their population peeks get offset due to randomness in the system.

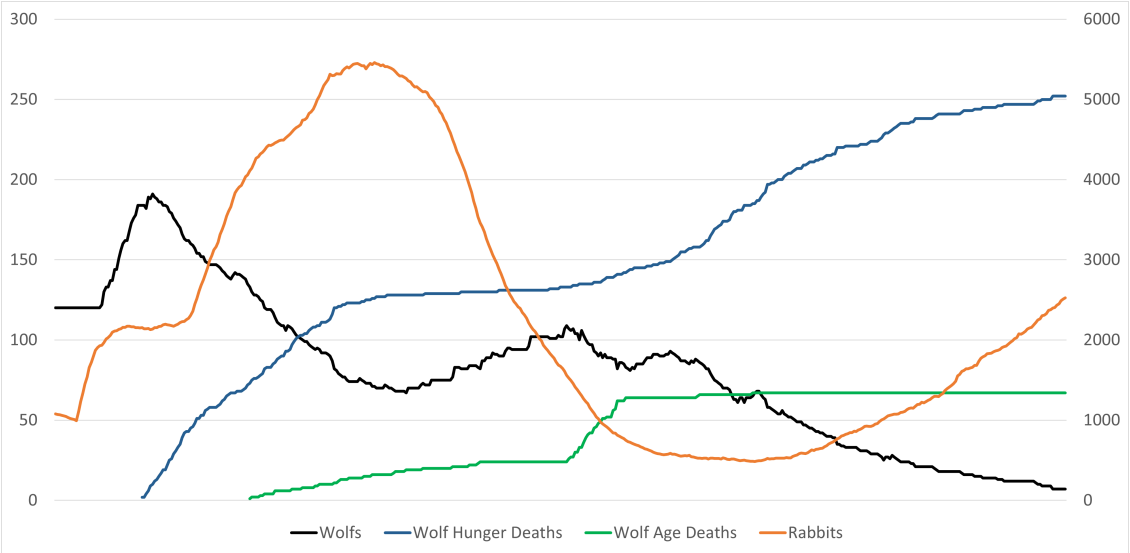

Figure 9.4: Graph displaying an outcome of a simulation where Wolfs were not resilient enough to last thought the population decrease of rabbits. Rabbits plot uses a secondary axis.

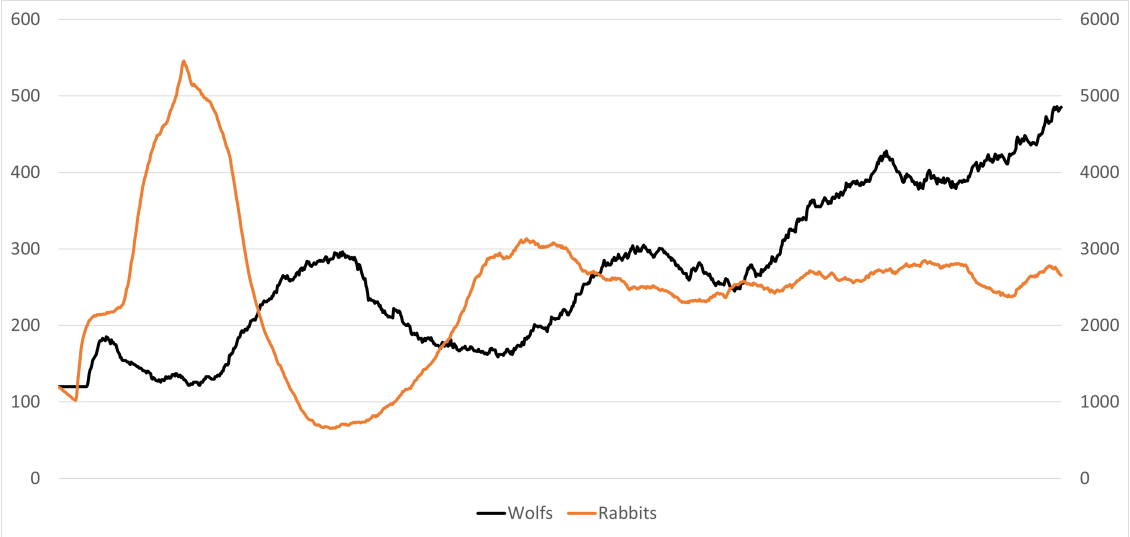

Figure 9.5: Wolf population growing without limit while leaving the rabbit population unaffected because enough food is generated by old wolfs dying. Rabbit plot uses a secondary axis.

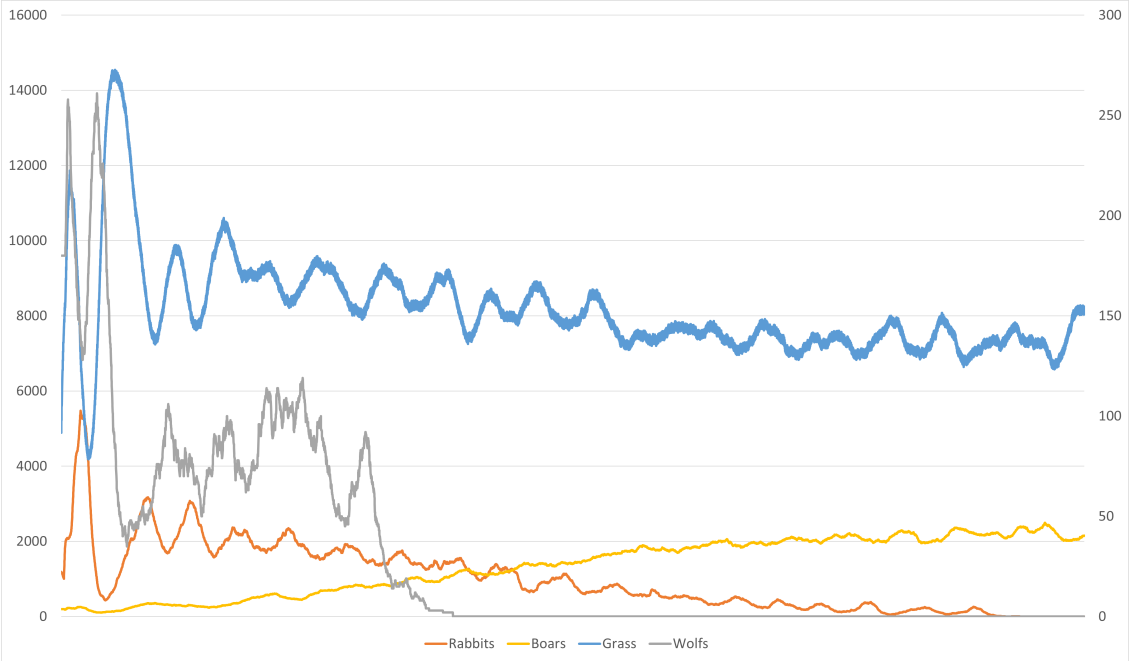

Figure 9.6: A two hour long simulation. Wolf plot uses secondary axis. This simulation took two hours. We can see that over the course of the simulation wolfs went extinct after about 45 minutes. It took over 90 minutes for rabbits to go extinct. This is due to the better mobility and higher tolerance for food deficiency of boars compared to rabbits.

Chapter 10

Conclusion

The goal of this bachelor thesis was to study entity component system architecture and use this architecture with the Unity engine. After getting to know the ECS paradigm I set out to create a simulation with thousands of agents which act based upon settings the user entered into the application. The target of the simulation was to create an ecosystem with different types of entities and accelerate that simulation. The last goal of this thesis was to measure the properties of the ecosystem and summarize the results into plots.

The final application uses a framework named Entitas as its ECS implementation. To keep the experience of the user pleasant the simulation has been accelerated by using pooling, time slicing and quad trees. This optimization allows for thousands of agents in the simulation. An interface provides the user the ability to enter initial simulation parameters and observe its progress.

This work also includes results of experimentation with the initial simulation settings and describes different problems encounter during this stage of development. Concluding with a two hour long simulation that demonstrates multiple phenomena that may be found in nature.

In the future this work might serve as a starting point for an interactive video game or can be expanded upon to include more detailed and realistic entity interactions.

Bibliography

[1] Abraham, S. Noise functions. *The University of Texas at Austin* [online]. The University of Texas at Austin, January 2021 [cit. 15-05-2021]. Available at: https://www.cs.utexas.edu/~theshark/courses/cs354/lectures/cs354-21.pdf.

[2] Baumel, E. Understanding data-oriented design for entity component systems. In: Game Developer Conference. *Data Oriented Tech Stack* [online]. Unity technologies, September 2019 [cit. 19-01-2021]. Available at: https://youtu.be/0_Byw9UMn9g.

[3] Bonet, R. T. Game Dev Guru. *CPU Slicing Secrets* [online]. Rubén Torres Bonet, February 2020 [cit. 12-05-2021]. Available at: https://thegamedev.guru/unity-performance/cpu-slicing-secrets/.

[4] Chakraborty, A. Tutorials Point. *Data Parallelism vs Task Parallelism* [online]. Tutorials Point, October 2019 [cit. 12-05-2021]. Available at: https://www.tutorialspoint.com/data-parallelism-vs-task-parallelism.

[5] Dmitry, M. Actors. *Github* [online]. Github, October 2018 [cit. 19-04-2021]. Available at: https://github.com/PixeyeHQ/actors.unity.

[6] Echterhoff, J. Incremental garbage collector. *Resources* [online]. Unity technologies, November 2018 [cit. 18-04-2021]. Available at: https://resources.unity.com/developer-tips/incremental-garbage-collector.

[7] EcsRx. *Github* [online]. Github, June 2016 [cit. 19-04-2021]. Available at: https://github.com/EcsRx/ecsrx.unity.

[8] What AI for Unity DOTS. *Pixelmatic.github* [online]. Github, May 2020 [cit. 19-04-2021]. Available at: https://pixelmatic.github.io/articles/2020/05/13/ecs-and-ai.html.

[9] Gregory, J. Memory Caching. *Naughty Dog Explains PS4's CPU, Memory and More in Detail and How They Can Make it "Run Really Fast"* [online]. DualShockers, . March 2014 [cit. 19-01-2021]. Available at: https://www.dualshockers.com/naughty-dog-explains-ps4s-cpu-memory-and-more-in-detail-and-how-they-can-make-them-run-really-fast/.

[10] Ilya, S. NanoECS. *Github* [online]. Github, December 2019 [cit. 19-04-2021]. Available at: https://github.com/SinyavtsevIlya/NanoECS.

[11] Leopotam. LeoECS. *Github* [online]. Github, January 2018 [cit. 19-04-2021]. Available at: https://github.com/Leopotam/ecs.

[12] Martin, A. What is an Entity System? *Entity Systems are the future of MMOG development – Part 2* [online]. T-machine.org, . November 2007 [cit. 20-01-2021]. Available at: http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/.

[13] Nikolov, S. OOP Is Dead, Long Live Data-oriented Design. *Youtube* [online]. CppCon 2018, . November 2018 [cit. 20-01-2021]. Available at: https://youtu.be/yy8jQgmhbAU.

[14] Value Noise and Procedural Patterns: Part 1 (Simple Pattern Examples). *Learn Computer Graphics From Scratch!* [online]. 2019 [cit. 19-01-2021]. Available at: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/procedural-patterns-noise-part-1/simple-pattern-examples.

[15] Oleg, M. MorpehECS. *Github* [online]. Github, Jabuary 2020 [cit. 19-04-2021]. Available at: https://github.com/scellecs/Morpeh.

[16] Rabin, S. *Game AI Pro : Collected Wisdom of Game AI Professionals*. 1st ed. CRC Press, 2013. ISBN 978-1-4665-6597-5,1466565977. Available at: http://www.gameaipro.com/.

[17] Schmid, S. Entitas. *Repository* [online]. Github, . March 2014 [cit. 8-04-2021]. Available at: https://github.com/sschmid/Entitas-CSharp.

[18] Sebastiano, M. Svelto Entity Component System. *Github* [online]. Github, January 2018 [cit. 19-04-2021]. Available at: https://github.com/sebas77/Svelto.ECS.

[19] Sebastiano, M. *Seba's lab* [online]. Sebastiano Mandalà, . 2021 [cit. 19-04-2021]. Available at: http://www.sebaslab.com/.

[20] Shaker, N., Togelius, J. and Nelson, M. J. *Procedural Content Generation in Games*. 1st ed. Cham Switzerland: Springer International Publishing, 2016. Computational Synthesis and Creative Systems, no. 1. ISBN 978-3-319-42714-0.

[21] Sharov, A. Lotka-Volterra Model. *The University of Texas at Austin* [online]. The University of Texas at Austin, December 1996 [cit. 20-04-2021]. Available at: https://web.ma.utexas.edu/users/davis/375/popecol/lec10/lotka.html.

[22] technologies, U. Understanding the managed heap. *Best Practices* [online]. Unity technologies, March 2018 [cit. 18-04-2021]. Available at: https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity4-1.html.

[23] technologies, U. Documentation. *Unity Tilemap* [online]. Unity technologies, May 2021 [cit. 15-05-2021]. Available at: https://docs.unity3d.com/Manual/class-Tilemap.html.

[24] Data Oriented Tech Stack. *Unity Entity Component System* [online]. Unity technologies, . 2021 [cit. 19-04-2021]. Available at: https://unity.com/dots.

[25] West, M. Trends in game programming: Emergent gameplay. *Game developer magazine*: *The Leading Game Industry Magazine* [magazine]. 1st ed., version 1.0. USA San Francisco: UBM Tech. September 2006, vol. 13, no. 8, p. 41–43. ISSN 1073-922X. Available at: https://www.gdcvault.com/gdmag.

[26] Wimble, K. ThinMatrix. *Equilinox* [online]. Karl Wimble, November 2018 [cit. 12-05-2021]. Available at: https://equilinox.com/.

[27] Zaks, M. Entitas Cookbook. *Cookbook* [online]. Github, July 2017 [cit. 15-04-2021]. Available at: https://github.com/mzaks/EntitasCookBook.

Appendix A

Media contents

- **text/** - Folder containing LaTeX source files
- **thesis.pdf** - Final thesis text build using LaTeX
- **src/** - Source files used by Unity to build the final application
- **build/** - Folder containing the resulting application build for Windows
- **2021-xkropa07-Simulation_in_Unity.mp4** - Video overview
- **readme.txt** - Text file with instructions on how to install the Unity Engine
- **data/** - Folder containing all the *.xml* and *.xls* data files used to create plots presented in this work