Universitetet
i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

BACHELOR'S THESIS

Study programme: Bachelor in Computer Science	Spring semester 2023 Open access
Author(s): Lukasz Pietkiewicz, Jakub Mroz	
Faculty supervisor: Erlend Tøssebro	
Thesis title: ECS and Optimization Patterns	
Course credits: 20	
Key words: ECS, optimization, game code patterns	Page count: 97 + appendix: 2 Stavanger 15. mai 2023

Contents

Abstract

OOP pattern has been overused in the game development world, so new architecture patterns like ECS are becoming more prominent.

This thesis aimed to research the Entity Component System programming paradigm, its practical uses and relevancy. We developed a 2D video game to try out ECS in a production scenario. The development of the product showed strengths and weaknesses of ECS, and its effect on development time, code quality and performance.

The research concluded that ECS provides gains in performance in many situations and scenarios. A well chosen ECS framework allowed for quick development time, as learning time of the ECS architecture was minimal. The ECS architecture allowed for decoupled code in a scenario with many systems and domains, which would demand more careful planning in other programming patterns.

Terminology

ECS - Entity Component System

EC - Entity Component - Component pattern used by most game engines

OOP - Object Oriented Programming - Programming paradigm based on the concept of objects

AoS - Array of Structures

SoA - Structure of Arrays

GC - Garbage Collection - Automatic memory management done by the framework

Mutable - Being able to change state

Boilerplate code - Code that is repeated in multiple places with little to no variation

Dead code - Code that never gets executed, or code that gets executed, but the results are never used

Singleton - A non-static object that there's a single instance of

CPU cycle time - Time for the cpu to complete all instructions

SIMD - Single instruction, multiple data

Lerp - Linear Interpolation - Interpolating between two or more values

Chapter 1

Introduction

1.1 Background

The idea behind this project started after seeing many video games on the market, especially relatively simple looking indie games (small games developed by an independent team), suffering from performance issues. That performance loss would often be most noticeable when too many entities would be present on the screen. Our goal was to find a way to develop video games which fixes the performance problem, or at least mitigates the issue.

Optimization is one of the most judged characteristics when it comes to video games, mostly measured in frames per second one can achieve while playing. When a video game is not properly optimized, it can exclude many people from being able to run it, and having an expensive high-end PC can become a requirement. An ideal situation would be to let as many devices to run the game as possible. It has happened multiple times in the video game industry that a game's reputation was damaged, because its optimization was suboptimal and many people who have bought the game could not get an enjoyable experience out of it.

After learning about the Entity Component System pattern, we thought that it can help in the optimization issue when developing video games [25]. It seemed to us that too many video games overuse Object-Oriented Programming (OOP) or Entity Component (EC) design, when Entity Component System pattern would be a better fit for their game, benefiting game performance, decreasing complexity, and maybe even decreasing production time of an end product.

1.2 Objectives

1.2.1 Inner workings of ECS and its practical use

We want to research how Entity-Component System works in theory and how it is put to work in practice. This will be done before and during development of the video game. This is necessary as ECS has vastly different system architecture than other programming patterns, so experience from them may not translate well.

1.2.2 Developing a video game

The developed video game will be a proper 2D roguelike genre video game, with permanent death and with focus on replayability and a varied upgrade system. The game should at least in part compete with other popular games of that genre existing today. We want our game to be enjoyable for roguelike players. To do that we will be focusing on developing a game that is easy to get into but hard to master, with multiple enemy waves and with genre-defining mechanics like player upgrades and randomness in each playthrough.

1.2.3 Relevancy of ECS

We want to compare ECS to other programming patterns to see their upsides and downsides. After that comparison we want to conclude whether ECS is worth investing time into by a game developer, or if it is better to stick to more traditional methods. When comparing, we want to look both at code clarity, development time and code execution times, to have a clear overview over upsides and downsides of ECS.

1.3 Project Description

We developed a game with focus on figuring out how to maximize the amount of diverse entities, mostly using ECS, locality of reference, branch prediction, sequentially structured data in memory. It was built on the Unity Engine with programming language C#. For ECS we used a library called DefaultECS [8]. The project is a 2D video game, like most of roguelike genre games today. It's a singleplayer game, because multiple players can be hard to fit in the roguelike genre experience, and multiplayer logic isn't a part of the thesis.

Roguelike is a genre of video games vaguely defined by permanent death, procedurally generated environment, limited resources, turn-based combat and randomness in each playthrough, although a roguelike game often does not have to meet all of those characteristics. Our game is going to skew towards the subgenre of “roguelites”, with no turn-based combat [19], and progress between new game runs. In our game the player finds himself in an arena, tasked with defeating enemies. Enemies come in waves, and with higher wave number enemies become more challenging both by numbers and game mechanics. The player is rewarded gold based on their performance in combat. They can then use it to upgrade their abilities to better deal with enemies. When the player dies, the run ends.

Chapter 2

Technologies and Tools

In this chapter we present the technologies used in our thesis, during development of video game and evaluation of ECS. We are going to explain what those technologies have been used for.

To research and find out more about optimization patterns and then develop a video game, we needed to settle on the technologies we'd use throughout the thesis. There were a few key important factors.

The first one was control. We needed to have enough control over the memory for it to be sequential when we wanted it to. So if a programming language only supported reference types, and didn't give any control on how the memory can be structured, then we couldn't choose it.

The second factor was speed, if the technologies we used had big overheads from the get-go, it would be harder to make a performant game and make good benchmarks.

The third factor was future set, we needed a particular set of features from ex. our programming language.

The fourth factor was ease of use. We didn't have time to learn a low level language like C++, it also wouldn't be smart to go too low level to languages like assembly for our programming language choice.

With these factors in mind, we decided on a few technologies.

2.1 C#

C# is used as a scripting language in our project. We have chosen it over other available alternatives, because it was a programming language we were already familiar with, and we would not have to use additional time on learning how to use a new language idiomatically. This meant we could use more time on learning ECS and other optimization patterns. C++ is often used in game development as well. It was an alternative choice to C# because it has a similar number of ECS libraries. In C# we would not have to worry about things like memory management, and for us C++ did not bring enough valuable benefits so we decided to proceed with C#.

C# was a great choice for a plethora of other reasons.

- It is a very general purpose language.
- It is a fairly modern language, with a lot of needed features (generics, lambdas, structs by reference, etc)
- It is a rather high level language, which makes it easier to write code. While still being performant enough to handle thousands of entities.
- It has been tested in many environments, and is very stable.
- It's currently the most popular language for developing video games.

But C# also has its downsides.

- Garbage Collection can be a problem if not handled correctly. Ex. without pooling, or high use of GC allocations.
- It isn't as bleeding edge as other new modern languages, like Rust.

2.2 Unity

Before we started creating a game, we first had to decide on which game engine are we going to use for our project. A game engine is an essential tool when creating a video game because it handles rendering, input, compiling to multiple platforms, etc... It can also simplify game development by providing other systems like physics and sound. Some of the most commercially popular game engines today are the Unity, Unreal Engine and Godot game engines.

Unreal Engine is not designed for creating small indie games, as it is more complex and more often used for developing AAA games, which means large budget games, produced and distributed by a major publisher. It is also less focused on supporting 2D games than the other two. Moreover it does not officially support WebGL, which means Unreal games cannot be ran in a browser. Unreal engine is free to use initially, but the developer has to pay royalty when their product sells for over 1 million U.S. dollars [3].

Godot on the other hand is smaller and less bloated than Unreal and Unity. It uses its own GDscript language, which is a wrapper code around C++ and has a python-like syntax. It is a relatively new game engine, so it is not as mature, reliable and tested as Unity and Unreal are. GDScript is a neat, easy language, with reference counting instead of GC or manual memory management. But it's very much lacking in features, compared to C#. Godot has a lot of platform integrations, including PC, mobile, Linux, web and macOS. Godot is completely free and a developer does not need to pay any royalties.

We chose Unity engine for our project because we decided it was best suited for our project. Unity natively supports C# as a scripting language [20]. Unity provides an implementation of a physics engine which calculates forces like gravity or collisions, although other engines have similar or same capabilities [23]. If sales of a Unity game exceed 100 thousand U.S dollars, then the developer has to switch from a free license to a paid subscription. Unity does have its own ECS framework [24], but it lacks proper documentation, and is still in the beta branch, which would make working with it very tedious as we would need to research its inner working to fully understand how to use it, so we decided to use a different 3rd party ECS framework instead.

2.3 Git

Git is an open source version control software. It enables synchronizing a project between multiple members and tracking of all the changes made, which makes working in a group easier. It has become an industry standard and is in wide use today by a wide variety of different companies, like Google or Microsoft [6].

2.4 Github

GitHub is a cloud storage service that stores git repositories. It also comes with a desktop version with GUI, which may be preferred over command line interface.

2.5 GIMP

For basic textures and sprites (images or art used in the game), we used GIMP. An open source GNU image manipulation program. To give us better feedback on how the mechanics work, and if correctly. Giving us a visual representation of the game states.

2.6 Audacity and Ableton Live 11

For the sound effect creation we have used two sound editors. Audacity is the simpler one, with better accessibility and work speed, but it lacks some features like more complicated audio effects. Ableton is not free like Audacity, it has a one-time license purchase. Ableton gives more tools for sound manipulation to the user, and is better when working with a large number of sound samples at the same time.

Chapter 3

Concepts

Since games are made in the same programming languages like other non-game software is made, they have a lot of similarities.

But because games as a product interact differently with their users, there's a few caveats and concepts that are more widely used in game development. We'll be looking at both, game programming concepts, and general software concepts.

3.1 Framerates and Optimization

In this chapter we give some background and explain concepts, which are important to understand what is the purpose behind video game optimization. We also explain a few CPU architecture concepts which will help to understand the inner workings of ECS later on.

3.1.1 Framerates

To achieve an appearance of motion, a film shows still pictures in quick succession. The same principle applies when creating motion effect on a digital display. Still pictures used in that process are called frames. Framerate describes how quickly image transitions happen and it is usually measured in frames per second (fps). In the film industry, 24 fps has been and still is the standard since the 1920s. It was seen as having good enough frame rate to be perceived as a fluid motion. If framerate drops too low, somewhere below 12-24 fps, a human eye starts to notice singular images instead of a fluid transition. 24 fps standard was a compromise to have as low frame rate as possible to reduce costs of analog film, since cost of using analog film increased with higher frame rates, i.e. 24 fps camera used half of analog film material as what 48 fps camera would use [12].

3.1.2 Perception by human eye

Even though 24 fps is enough to create an appearance of motion, human eye can perceive higher framerates as noticeably smoother. Right now, the industry standard for a computer monitor refresh rate is 60Hz, meaning it can show content that is up to 60 fps. In "Evidence that Viewers Prefer Higher Frame Rate Film" from 2015, researchers find:

> (...) there was a clear preference for higher frame rates (48 and 60 fps) when contrasted with a standard of 24fps, regardless of content. It is clear from these data that, overall, viewers prefer higher frame rate footage, and that there is a significant impact on preferences of increasing frame rate from 24 to 48 fps [2].

Same research says that with higher fps number, the difference for a person watching becomes smaller, but there is still a difference. Today there are commercially available products which can display 120, 144 or 240 fps.

3.1.3 Optimization

Optimization is a process of improving performance, which in a game context means reducing single frame creation time. 60 fps is the go-to number a game should reach to be considered playable. The goal for a game developer is to make their game run 60 fps on as many devices as it is possible, to increase their potential customer base. In the past, the goal framerate used to be 30 in some parts of the video game market, especially in older consoles like PS3 or Xbox 360.

To achieve that framerate, a single frame calculation cannot exceed 16.67ms on any component (CPU or GPU). Even if a game can hit 60 fps on most devices, there are still valid reasons to optimize a game further, like reduced power consumption and more "backup power" if someone would want to run a game on a framerate higher than 60.

3.2 CPU Cache

When data flows between CPU, disk, and RAM, the CPU can try to preload data on which it will do instructions on in the future, by also grabbing the data next in memory to the currently loaded data. This data will be stored in one of the CPU caches: L1, L2 or L3.

Why did CPU cache become a thing? With electronic technology improving in general, some technologies improved faster than others. This is especially the case when comparing CPU speeds against time it takes to send data to CPU from RAM or disk. The speeds between these operations are:

Cache Type	Latency
L1	1ns
L2	4ns
L3	40ns
RAM	100ns
Disk	50µs

[7]

There are three CPU caches, L1, L2, and L3. L1 is the smallest one, and the fastest one. It's meant for the constantly used data, that the CPU is doing work most often on. Most modern CPU's include up to 512KiB L1 cache (64KiB per core) [1], they're usually split into two cache types. First one being a L1 instruction cache, and the other is a L1 data cache. Each core has it's own L1 and L2 caches, while the L3 cache is shared between all of them.

Most CPUs have use a range between 4 and 8 MiB for the L2 cache size, while the L3 varies in the 10MiB to 64MiB ranges.

3.2.1 Cache hit rate

A cache hit happens when data requested by the CPU is already present in the cache and does not need to be loaded from system memory. A cache hit rate describes how many cache hits there are in relation to all the data requests done by a CPU. With a bigger cache hit rate (described as a percentage), the less time the CPU has to spend to wait for the data to be loaded from memory [10].

3.2.2 Locality of Reference

The main types of localities for ECS are temporal, spatial and branch localities.

Spatial locality, is about data being sequentially together in memory, causing the CPU cache to contain the correct next data. This improves the iteration process

Branch locality, is also very relevant to ECS, because branching inside our systems is very minimal, but when it's done it's a very rare side effect.

3.3 Branching

Branching is about CPU's branch predictor, when it tries to guess which way a branch (example: an if-else statement) will go before it's known. If the prediction was correct, the speed of the process is greatly increased. If on the other hand it was wrong, the CPU goes back and executes the other correct branch, making a slight delay.

The more times a CPU gets a decision between branches, the more information it gets on the branch, the more accurate the branch prediction will become. Further increasing the performance.

Branching as a whole can be entirely avoided, if the code doesn't have any conditional statements. This can sometimes be achieved my using math and bit functionality, instead of conditionals.

Chapter 4

ECS

ECS - Entity Component System is a very basic concept born from the need of a new way of thinking for programming games and some real-time applications.

Game programmers have noticed that classic OOP patterns, like inheritance, don't work well in games. And had to find their own collection of common software design patterns to follow.

4.1 ECS Functionality

ECS consists of three parts.

- Entities - Identifiers (ID) for sets of components - Ex. player and enemy entities
- Components - Data - Ex. Velocity and position components
- Systems - Logic that does work on components - Ex. Velocity and position systems

ECS frameworks usually also have a "World" object. The world object stores all of the entities and components.

One of the main strengths of ECS is how the data and logic is structured, and how they interact with each other. ECS guides the programmer to think in terms of data, and how it is processed. This is in contrast to OOP, where the programmer thinks in terms of objects, and how they interact with each other. In ECS, the data has no inherently tied logic. The logic is defined by systems instead.

In general ECS makes it harder to write tightly coupled code, this is because like mentioned before, the data and logic are separated.

The beauty of ECS is being able to have multiple entities reuse the same components and logic. Example: All projectiles and enemies move physically in the game world. So they all use the same position and velocity components and systems.

But ECS does have its own problems. A big one is duplication. A simple example is a timer component. A timer component has a numeric value, that is increased or decreased by a system. And now let's say that we want to do two actions every time the timer reaches 5 and 0. They both want to reset the timer after their action. But this would disrupt the other action. So the solution would be to have two timers, one for each action. Usually in the same action component.

This could be solved by adding extra complexity like events, but this would in general spiral out of control, complexity-wise.

World

Most ECS frameworks use objects called "worlds", they're responsible for storing all the entity and component data.

Worlds are usually used to separate contexts or domains, like main menu, options menu, gameplay scene, etc. This allows for elegant control of separate logic structures, without the need of programming side effects, or worrying about different contexts colliding.

Worlds are also used as the main reference object, as in the object that we use to interact with the entire framework.

Entities

There's two approaches to how entities look like, the most common one, is a struct with three IDs: WorldId, EntityId, and GenerationValue. In the vast majority of ECS frameworks, the entities won't have more data fields than these three. One of the core reasons behind this, is to fit an entity into 64 bits. Most frameworks do this by having WorldId and GenerationValue be 16 bit (short), and EntityId be 32 bit. It's important to fit these entities into 64 bits because of how CPU registers work. 64 bit architecture CPUs have 64 bit registers, making them a perfect fit for the entities, without any padding and extra space.

WorldId tells the systems to which world the entity belongs to, so we don't mix entities in different worlds. Example: gameplay world, and hub world.

EntityId is the id that the systems use to identify which components belong to the entity.

GenerationValue increments every time we make a new entity with the same EntityId, in case the previous entity was destroyed. This is used to differentiate between new and old entity versions, in case we saved an old entity reference somewhere, and it doesn't exist anymore.

The other less common way of entity design, is to have the public API give entities as singular integer values. While still using the classic entities in the backend. This can be done to maximize the performance of ECS. The biggest issue with this approach, is that it forces the developer to write a lot of boilerplate code. Making the entire development process longer for each new system. So the vast majority of frameworks don't do this, one exception being LeoEcsLite [17].

Components

As mentioned before, components can be classes or structs, and sometimes even interfaces. But the majority of frameworks only support structs, or treat them as first-class citizens [18].

The reason for that being is that structs are value types, as opposite to class and interface types, being reference types. And as mentioned before, in section 3.2 cpu cache hit rate is important to maximize performance of SoA setups. And reference types would ruin that hit rate. Because the memory that's allocated internally for these reference types might not be sequential. But using interfaces as components can be beneficial and can be a solution to code design issues. That being said using interfaces with structs introduces a new potential performance problem, boxing and unboxing.

Components can be of two functioning types, data or flag.

Data components mostly contain mutable data, that's worked on by systems. Examples of this are: Position, Velocity, Health.

Flag components don't have any data, and are treated as indicators. Examples of this are: IsPlayer, IsFlying, IsEnemy.

Because the components are usually structs, the language has to either support pointers, or passing value-types by reference. This is very important, because without it, we wouldn't be able to pass structs to functions without copying them each time. We would be forced to pass an array/vector with the index that needs to be updated. This would complicate already complex systems to a point of insanity. This would also mean every time we want to use the components data multiple times, or update it and use it. We would be forced to either get the component from the array each time, or copy the component in the beginning and set it back updated in the array in the end. Both options being terrible performance-wise.

Systems

The general purpose of systems is to do work over components. Systems should be responsible for almost all logic in ECS.

Almost all systems in ECS work on filters/queries, this way we can control what logic applies to which entity. It's usually done in the beginning of an ECS system with attributes, or in the constructor.

```
[With(typeof(IsEnemy), typeof(VelocityComp))]
[Without(typeof(IsChargeEnemy), typeof(IsSleeping))]
```

Code 4.1: System filter through attributes

```
world.GetEntities().With(typeof(IsEnemy), ...
    typeof(VelocityComp)).Without(typeof(IsChargeEnemy), ...
    typeof(IsSleeping)).AsSet();
```

Code 4.2: System filter through constructor

In this example we select entities which have both IsEnemy and VelocityComp components, but they cannot have any of the components excluded in “Without” filter, ex. enemies that have a special charge movement, or are sleeping, and shouldn't move. After selecting entities that we wanted, we can then start performing calculations on them inside the system.

In our project we have used two types of systems. The first type is as in the example above, which operates on a given set of entities. The other type does not work on a set of entities, but instead they handle things like state of the game, inputs of a player or the options menu. An example of this type of system is the WaveSystem, which manages switching to next wave and logic behind it. Its job is to transition to a next wave at the right time, increment wave number and start a new wave. It never has to work on any entity set, so it does not have any entity filtering.

We're also able to work on multiple entity sets in a system, this is especially useful when multiple entity types need to get information about one another.

For example we need all the enemy entities in our automatic targeting system, but we want to iterate it over once, for the player. So what we do is we iterate over the player, and make an enemy entity set in constructor.

```
[With(typeof(IsPlayerComp))]
class AutomaticTargetingSystem : AEntitySetSystem<float>
{
    readonly EntitySet _enemyEntitySet;

    public AutomaticTargetingSystem(World world) : base(world)
    {
        _enemyEntitySet = world.GetEntities().With<IsEnemy>().AsSet();
    }

    protected override void Update(float state, in Entity entity)
    {
        foreach (ref readonly var enemy in _enemyEntitySet.GetEntities())
        {
            //Automatic targeting logic
        }
    }
}
```

Code 4.3: Player automatic targeting ECS system

4.1.1 ECS Types

Sparse set

Sparse set based ECS is the easiest ECS type to work with. And the easiest to understand.

In sparse set ECS all component types are stored in single arrays (almost always SoA - Structure of Arrays).

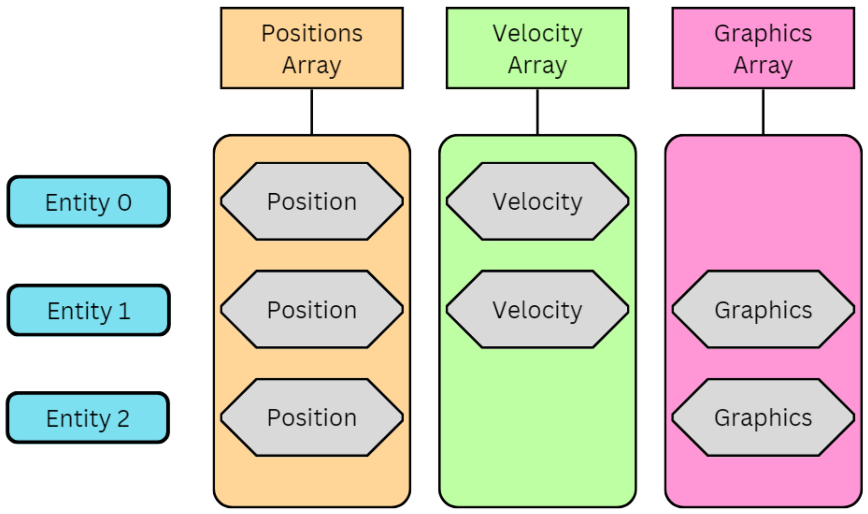


Figure 4.1: Sparse set arrays

In this example image, entity 0 can be a moving trigger, entity 1 be a player, and entity 2 a stationary enemy.

The biggest advantages of sparse set ECS is fast adding and removing of components on entities, and is very easy to work with. While the biggest disadvantage is that is has the slowest iteration compared to other ECS types.

Archetype

Archetype based ECS is the fastest ECS type for iteration. This is because of how the components are being stored.

Instead of having filters for systems that sort out which entities should be handled, it uses arrays of packed components. One for each unique components count.

So if we make two entities: Player, with input, position and health components. And enemy, with position and health components.

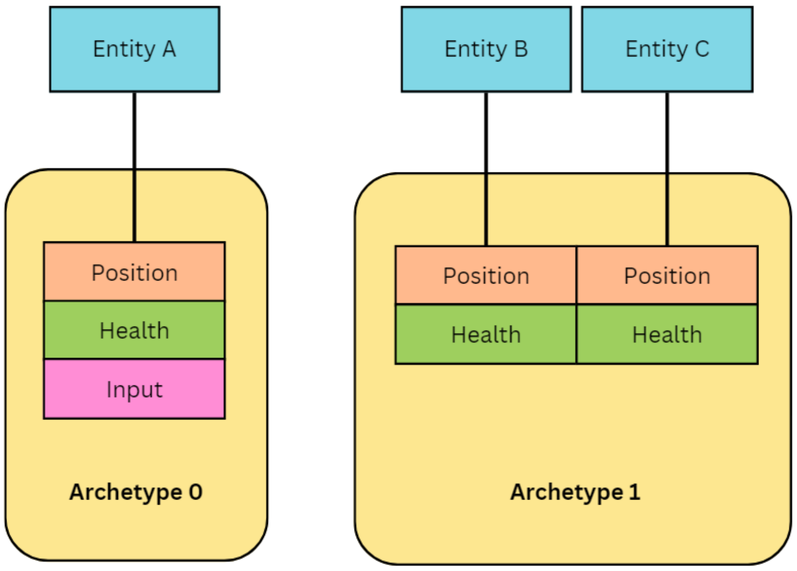


Figure 4.2: Archetype set arrays

In sparse set, these same components types would be stored together. But in archetype ecs, they're instead stored separately by unique sets of components.

This way we'll have access to all the archetype components at once.

The biggest challenge with archetype is adding and removing components. Since when we add or remove a component, the entity changes their internal components set, so we'll have to move all it's components to a different archetype.

Archetype ECS introduces more complexity to the ecosystem, and requires much more careful design. It should be used for the most part by developers who know ECS well, how to develop around it, and who know archetype ECS is the right fit for the project.

4.1.2 Systems Communication

Systems have to share information between each other. The question is how to do it in an elegant way.

In an ideal, perfectly programmed game, the majority of systems would be isolated. The systems wouldn't communicate with each other. This is very hard/next to impossible to achieve, so we have to find a way to communicate between systems in a graceful manner.

There's four ways for systems to communicate with each other

- Short time (one frame) components - Components that exist for a very short time, usually one CPU update cycle.
- Events - Signals sent between the world/systems
- Merging systems together instead - this is the best way for performance-sensitive contexts. But will increase the complexity of the system exponentially. Also makes the system non-modular.
- World/static components

One Frame Components

One frame components is a very neat type of communication, because components are native to ECS. But right away the biggest drawback is that one frame components destroy filter updates in sparse set ECS, and are impossible to implement well in archetype ECS. If we need hundreds of signals, this becomes catastrophic performance wise. And therefore is the worst solution when it comes to performance.

Events

To aid the communication between systems and/or the world, some ECS frameworks use events. Events are very powerful in ECS, but they can be misused very easily as well. Events should be unique, in the sense that every subscription to an event should be logically clear.

A bad subscription is a subscription that doesn't have a logical connection with the publisher, or is a hitch subscriber. Meaning that the connection is faulty. An example on this would be: Let's imagine a game over system. It fires an event called "GameOverEvent" when the player dies. Now we have a few systems that want to subscribe to that event: "PlayDeathSound" by "DeathSystem", "ShowGameOverMenu" by "UISystem", and "SubmitGameScore" by "ScoreSystem". If any of these separate subscribers depend on each other, it's most likely a bad subscription. Instead, all of them should be subscribed to our original "GameOverEvent" and not care or know about each other. Note that the order of execution can be important here as well. If "ShowGameOverMenu" displays the score on a leaderboard, it will need data from the "SubmitGameScore" system. But that update feature should probably be handled with a new event, fired by "ScoreSystem", when the score got submitted.

That being said, events are usually the best solution to solving inter-system communication.

Merging Systems

Merging systems is the least used solution, but also the fastest one. Sending over 1000 events/1 big event or making 1000 new one frame components will always be slower than having the systems work together as one. Unfortunately this approach is the least elegant one. It should only be used in hot paths/performance sensitive cases, or when communication of two systems raises cpu time drastically.

World/static components

DefaultECS has a feature where components can be stored directly on the world, instead of on an entity. This is another great way to handle shared state between systems.

Other frameworks achieve this with an injected service, that contains all the shared info between systems. This works worse than world components since it introduces another logic layer to ECS.

Our Choices

We used events in the majority of systems and controllers communication. This is because it was the cleanest choice.

We did also use world components in some specific cases where state was needed to be saved between multiple systems.

```
world.SetSingletonInstance(ScoreComp.Default);
world.SetSingletonInstance<GoldComp>();
world.SetSingletonInstance<KillStreakComp>();
world.SetSingletonInstance(UpgradesComp.Default);
world.SetSingletonInstance(EnemyUpgradesComp.Default);
world.SetSingletonInstance(new SpawnAreaComp { Value = new ...
    SpawnArea(bounds) });
world.SetSingletonInstance<WaveInfoComp>();
```

Code 4.4: Singleton world components

Since DefaultECS allows for max capacity of components, we made a extension method to make singleton instances.

```
public static void SetSingletonInstance<T>(this World world, T instance ...
    = default) where T : struct
{
    world.SetMaxCapacity<T>(1);
    world.Set(instance);
}
```

Code 4.5: Singleton world component extension method

This could have also been achieved by having these components on the player (since there's only one in the game). But this would lead to a lot of complications if we wanted to have more than one player to the game. So having it on the world is a safer bet, and it doesn't introduce technical debt. It would also force the systems to make and use the player filter each time we want to access these components.

4.1.3 SoA and AoS

SoA (struct of arrays) and AoS (array of structs) are two different ways to store data. Each having their upsides and downsides.

SoA

In SoA a struct contains all the arrays, this approach is much better when it comes to multi-threading and cpu cache hit rate. This is also the way ECS stores components, sequentially beside each other. It's also better suited for vectorization and SIMD instructions.

SoA is almost always the more performative option of the two, so it's more suited for performance-critical contexts. But SoA is also harder to work with, since all the data parts are split between multiple structs.

AoS

AoS is a more OOP oriented approach. And much more intuitive to developers. Since the data is usually stored together in objects, instead of split into components/parts. All values can be read and wrote to from one object.

AoS can be a better choice when performance doesn't matter, and the developers are used to OOP. Since AoS is often easier to comprehend and work with.

4.1.4 Concurrency

Concurrency can be a great way to get extra performance without refactoring systems. It achieves this by running logic at the same time, or independently splitting up tasks. This usually comes at a cost though, of either increased complexity. Or forcing the developer to restructure their systems for parallelism to be viable or even doable.

One of the other main strengths of ECS, is that it data is one specific place. And is being worked on by systems independently. So we can utilize concurrency and parallelism very easily. Because our logic doesn't have any side-effects. So having parallelism inside systems is a great solution if we need extra performance.

Parallelism of multiple systems together on the other hand is much trickier, because the components data state matters to some systems. Order of systems matters as well, forcing us to think very carefully about how to run systems in parallel and which, if any.

The simplest ones to figure out are independent systems, systems that don't care about component state changes, or are the only ones that care about these state changes. The issue is that these systems usually aren't the performance bottlenecks.

This type of concurrency would be very hard to implement in a classic OOP game project. It is because we would have a lot of side effect state changes, forcing us to either redesign how they're using and mutating state. Or implement locks in key places, further increasing the complexity of the application. This isn't the case with separated systems logic, like in ECS.

4.2 ECS Frameworks

We decided not to create our own ECS framework. This would take too much time away from the game development process. The process of creating the ECS framework would be very demanding, as we would need to test many scenarios to be sure it worked properly and many mechanics like linking entities to components, or filtering entities, are very hard to implement performatively. Potential problems could slow us to much to finish the rest of the thesis, especially if those problems would not be found in time. This is why we decided that making our own ECS framework wasn't worth it.

We decided to use DefaultECS [8]. We had a few requirements for the ECS framework:

- Lightweight
- Sparse-set
- Struct-based components
- Minimal/no dependencies
- World events/message (that aren't one frame components)
- Open Source
- Has documentation
- Not reactive-based system logic
- Not code generation based

Optional, but nice to have:

- Components can live on a World
- Unit tested
- Unity Editor support (runtime editor debug view)
- Enabling/disabling components and entities

The alternatives were:

- UnityEntities [22]
- morpeh [16]
- LeoEcsLite [17]
- Svelto.ECS [11]

- DefaultEcs [8]

Unity Entities didn't fit most of our requirements.

Morpeh had it's entire source code in one file, and seemed poorly designed.

LeoEcsLite was very lightweight, but it was missing too many important features.

Svelto.ECS had good predispositions, but it was too complex, and would take too much time to fully learn.

And in the end, DefaultECS fits all of our requirements, even most of our optional ones.

A comparison table one of us made before, helped us identifying which ECS will suit our needs [18].

Requirements explanations

Sparse set: Since this was our first ECS project, sparse set ECS was our best option. Balanced learning curve, adequate performance, and allows for elegant design.

Struct based components: One of major points of ECS is performance; cpu cache lines. If we used classes instead, we would destroy cpu cache hit rate.

World events/messages: Best way to communicate between systems is events. So we'd either have to have find an ECS framework with it, or make our own events logic.

Open source: We had a feeling we might need to modify the ECS framework more to our needs. It would also be nice to be able to research how logic works in the backend.

Not reactive-based system logic: Reactive ECS uses messages/signals resulting from entity component changes to keep track of which entities match systems/queries. Reactive ECS is much slower than non-reactive [9].

Not code generation based: Code generation has quirks, and introduces a middle man to the code. Making it harder to work with the source code if needed.

4.3 ECS compared to other programming paradigms

4.3.1 ECS compared to OOP

ECS pattern follows the KISS (keep it simple, stupid!) principle a lot. There's no need for dependency injection, service pattern, subtype pattern, and other patterns that were made from OOP principles.

ECS almost exclusively only uses:

- Entities for pointing towards a set of components
- Components for storing data
- Systems for logic
- Events for messages between systems and controllers

And that's pretty much it complexity wise. Everything is a mix of these patterns/principles. The idea is close to how to the programming language GO was designed, with simplicity in mind, limiting the amount of keywords, and making sure that things can be for the most part done in one way. The language most well known for being able to do something in too many ways is Ruby.

Health Component, is a generic component used for both our player entity, and enemy entities. In OOP pattern we would need to either make a generic class/struct anyway for storing current and max health. Or create two logic systems, one for player, one for enemy. Or create one base system, and then two systems that inherit it.

Using polymorphism can often lead to many problems in the future. An example for that would be if we wanted to introduce new types of enemies, or allies. The new types will be bound by the original abstract class.

This is where ECS shines. We can distinguish between entities in two ways. By making two systems, one that handles all entities with HealthComponent and IsPlayerComponent. Or checking for IsPlayerComponent flag inside the main system.

Position component is another perfect example for this.

We'll still be using the component programming pattern [14][15]. Because using classic OOP is not the best idea for game programming. We'll look into why later.

EC

Let's look at the PositionComponent from an Entity Component (EC) perspective. In OOP there's two approaches, either make every physical entity have it's own float X, Y. Then have scripts that work on these values with their own logic. Or encapsulate part of the logic into another script, and try to reuse it in multiple entities, like enemies. The second option is how EC works.

Ex. player has a "PlayerMovement" script, that gets input, checks for bounds/boundaries, then applies that input with player speed to player velocity.

Now here we'll have code duplication, because both enemies and player use the concept of velocity. So we might want to encapsulate that as well. But what should we do when the velocity needs to be handled differently between player and enemy? Do we now create another abstraction for velocity that we feed to both entities? Ok, that works. But now we want some enemies to move differently, or maybe not move at all. Right away, we're running into problems here. Note, they can be solved, but it will sooner or later lead to some inelegant code, unless we refactor big parts of the systems each time we introduce new features like these.

This is how a EC based approach would look like.

```
sealed class PositionComponent
{
    public float X;
    public float Y;
}

sealed class VelocityComponent
{
    public float X;
    public float Y;
    public float Speed;
}

interface IMovementSystem
{
    void Update(float ∆Time);
}

sealed class VelocitySystem : IMovementSystem
{
    readonly PositionComponent _position;
    readonly VelocityComponent _velocity;

    public void Update(float ∆Time)
    {
        _position.X += _velocity.X * _velocity.Speed * ΔTime;
        _position.Y += _velocity.Y * _velocity.Speed * ΔTime;
    }
}

sealed class BoundsVelocitySystems : IMovementSystem
{
    readonly PositionComponent _position;
    readonly VelocityComponent _velocity;
    readonly Bounds _bounds;

    public void Update(float ΔTime)
    {
        if (_position.X < _bounds.Min.x && _velocity.X < 0 ||
            _position.X > _bounds.Max.x && _velocity.X > 0)
            _velocity.X = 0;
        if (_position.Y < _bounds.Min.y && _velocity.Y < 0 ||
            _position.Y > _bounds.Max.y && _velocity.Y > 0)
            _velocity.Y = 0;

        _position.X += _velocity.X * _velocity.Speed * ΔTime;
        _position.Y += _velocity.Y * _velocity.Speed * ΔTime;
    }
}

sealed class PlayerEntity
{
    readonly IMovementSystem _movementSystem;
    //Player specific fields

    public void Update(float ΔTime)
    {
        _movementSystem.Update(ΔTime);
    }
}

sealed class EnemyEntity
{
    readonly IMovementSystem _movementSystem;
    //Enemy specific fields

    public void Update(float ΔTime)
    {
        _movementSystem.Update(ΔTime);
    }
}
```

Code 4.6: Example EC movement code

ECS

Now let's look at the same logic, but in ECS. Player entity has: input, position, velocity components. World entity has: bounds component.

Player input systems can't be used by enemies, so it's unique for the player.

```
[With(typeof(IsPlayerComp))]
class InputSystem : AEntitySetSystem<float>
{
    protected override void Update(float ΔTime, in Entity entity)
    {
        float horizontalAxis = Input.GetAxis("Horizontal");
        float verticalAxis = Input.GetAxis("Vertical");

        if (horizontalAxis == 0 && verticalAxis == 0)
        {
            entity.Get<VelocityComp>().Direction = Vector2.zero;
            return;
        }

        var direction = new Vector2(horizontalAxis, ...
            verticalAxis).normalized;

        entity.Get<VelocityComp>().Direction = direction;
    }
}
```

Code 4.7: ECS input system

In the project, we have a custom player movement system. That's because we want the player's velocity to be lerped and smooth. But this doesn't need to be the case.

```
[With(typeof(VelocityComp))]
class VelocitySystem : AEntitySetSystem<float>
{
    protected override void Update(float ΔTime, in Entity entity)
    {
        ref readonly var velocity = ref entity.Get<VelocityComp>();

        entity.Get<PositionComp>().Value += velocity.Direction * ...
            (velocity.Speed * ΔTime);
    }
}
```

Code 4.8: ECS velocity system

Classic OOP

The classic OOP ideas and patterns tell us to use polymorphism to combat code duplication, and this can work well in many programming scenarios. But not in games.

This becomes a terrible approach, when we later on want to add new types of movement, like "jumping movement", or "diagonal only". We want most of them to respect the boundaries, but not all. So we make new slightly different versions (code duplication). Or try to force the base classes to follow the rules with flags or some other way. Which leads to terrible design and dead code.

```
interface IMovable
{
    void Move(float ∆Time);
}

abstract class WorldObject
{
    protected float positionX, positionY;
}

abstract class MovingObject : WorldObject, IMovable
{
    protected float velocityX, velocityY;
    protected float speed;

    public virtual void Move(float ∆Time)
    {
        positionX += velocityX * speed * ∆Time;
        positionY += velocityY * speed * ∆Time;
    }
}

abstract class BoundedObject : MovingObject
{
    readonly Bounds _bounds;

    public override void Move(float ∆Time)
    {
        if (positionX < _bounds.Min.x && velocityX < 0 ||
            positionX > _bounds.Max.x && velocityX > 0)
            velocityX = 0;
        if (positionY < _bounds.Min.y && velocityY < 0 ||
            positionY > _bounds.Max.y && velocityY > 0)
            velocityY = 0;

        base.Move(∆Time);
    }
}

```

```
sealed class Player : BoundedObject
{
    //Player specific fields
}

sealed class Enemy : MovingObject
{
    //Enemy specific fields
}
```

Code 4.9: Example OOP movement code

4.3.2 ECS compared to Unity EC

Unity uses the Entity Component (EC) pattern. The main difference between EC and ECS, is that the logic in ECS is inside systems, while the logic in EC is inside components.

EC pattern is a very easy way to write code in. But without masterful planning, it will lead to a tangled web of connections.

Let's compare Entity Component in Unity against ECS in practice. Here is a comparison of two simple scripts which make enemy follow the player in a game.

Here is the EC version:

```
1  public class FollowPlayer
2  {
3      [SerializeField]
4      Transform player;
5      [SerializeField]
6      Transform enemy;
7      [SerializeField]
8      Rigidbody2D rb;
9      float movespeed = 5f;
10     Vector2 moveDirection;
11
12     void Update()
13     {
14         moveDirection = (player.position - enemy.position).normalized;
15     }
16
17     void FixedUpdate()
18     {
19         rb.velocity = new Vector2(moveDirection.x, moveDirection.y) * ...
               movespeed;
20     }
21 }
```

Code 4.10: Example Unity-based movement component

In Unity EC most of game's content is represented as Components parented to GameObjects. This script would be attached as Component to Enemy GameObject in Unity. It means there would be one script of this kind running for every Enemy instance created. It could potentially reduce performance if too many such scripts ran in the background. On the other hand Unity EC has a benefit of being a very modular system and is relatively easy to write code in, although it may not be easy to keep the code clean and simple.

Player transform and Enemy's own transform fields in lines 4 and 6 are being fed into the script by serializing them into Unity, and then manually choosing which components go in by drag-and-dropping them.

Update and FixedUpdate functions are the hot path, and are calculated every frame cycle. FixedUpdate is used for physics calculation and has fixed delta time between calculations, usually 20 milliseconds. Inside Update function we calculate movement vector by substracting player position vector from enemy's own position vector. Then we apply this vector into Rigidbody Component of enemy. Rigidbody's function is to react to physics and apply motion to parent GameObject.

Let's now look instead at an ECS script that achieves the same goal:

```
1  [With(typeof(IsEnemy), typeof(VelocityComp))]
2  class EnemyFollowSystem : CustomEntitySetSystem
3  {
4      protected override void Update(float ∆Time, ReadOnlySpan<Entity> ...
           entities)
5      {
6          var playerPosition = Player.Get<PositionComp>().Value;
7  
8          for (int i = 0; i < entities.Length; i++)
9          {
10             ref readonly var entity = ref entities[i];
11             ref readonly var position = ref entity.Get<PositionComp>();
12 
13             var normalized = (playerPosition - position.Value).normalized;
14             entity.Get<VelocityComp>().Direction = normalized;
15         }
16     }
17 }
```

Code 4.11: Enemy follow ECS system

In the first line we select which entities get selected for this script. In this case we want entities who have both IsEnemy flag and Velocity component, which allows an entity to move.

Then inside the Update function, which is the hot path of the script, we read player position component in line 6 before entering a for loop that spans all selected entities from line 1. This is because all entities will use the same player position in their calculations. ECS components are not the same as Unity Components, and they are not serializable into Unity by default.

Inside for loop we read enemy's own position and with that we can calculate movement vector for the entity, which is finally written into enemy's velocity component.

There is only one instance of this script running in the core of ECS. Compared to N number of components for each enemy if we used EC.

Chapter 5

Game systems and content creation process

In this chapter we explain how our game was developed, and how important mechanics were designed and implemented. On some smaller features, we will only focus on the design. Showing how game systems are implemented will visualize how ECS is put into a production scenario.

Code snippets shown in this chapter are often simplified versions of the final product, in order to show basic logic and function of a system as they were created. Later on in the development the systems had to hold more logic, as more content was added to the game.

5.1 Basics

At first we started with the bare-bones skeleton of the project, the absolute basic MVP (Minimum Viable Product). In the MVP a player could move, enemies would spawn, and follow the player, and that's it. We started work in unity by preparing a gameplay scene and a simple 2D background for it.

5.2 Player

Then we started working on creating a playable player character. We wanted the player character to be an object that is in the centre of the screen. A player will then be able to control movement of the player character on a 2D plane by using WASD controls. This means that the player character can be moved on the horizontal and vertical axis. Camera will follow the player character so that it stays in the middle of a screen. With visuals being shown by a Sprite Renderer component on a Unity GameObject.

Figure 5.1: Player character entity

5.2.1 Player Entity

The player entity will be spawned by a PlayerSpawner class in the GameController. In there, the player will receive all necessary components. Relevant to us now are:

- GameObjectComp: ties a Unity GameObject with ECS's Player Entity
- RigidbodyComp: holds a reference to RigidBody from the Unity GameObject, which is used in calculating movement
- VelocityComp: which carries values for movement speed and direction

Fields inside VelocityComp:

```
public float Speed;
public Vector2 Direction;
```

Code 5.1: Velocity component fields

5.2.2 Input System

The first mechanic we needed to create, was a way for the player to control the movement of the player character with input capture. Unity has its own input capturing system and we decided to use it. We created InputSystem ECS system which will hold the input logic.

```
float horizontalAxis = Input.GetAxis("Horizontal");
float verticalAxis = Input.GetAxis("Vertical");
```

Code 5.2: Player movement input capture

In this code fragment inside InputSystem we use “Horizontal” axis, which is binded to keys A and D by default, and “Vertical” which uses W and S. The way it works is that if the A key is pressed it changes the axis value to -1, and the D key changes value to 1. Both change the value linearly, with a smooth motions between 0 and the destination value. If none or both keys on the same axis are pressed, the value becomes 0. Unity editor allows to change what keys correspond to which action in those axes, but we used default values.

```
var direction = new Vector2(horizontalAxis, verticalAxis).normalized;
entity.Get<VelocityComp>().Direction = direction;
```

Code 5.3: Code for calculating velocity direction

In these two lines we take horizontal and vertical input of a player, normalize it and save the values as a Vector2. Vector2 is a custom Unity structure used to represent 2D vectors. Then we store that Vector2 value inside player character’s VelocityComp, so that it can be read by the VelocitySystem. The PlayerMovementSystem will later use the info from VelocityComp and move the player entity towards the direction read from VelocityComp’s value.

5.2.3 Player Movement System

To control the movement of a player we have a PlayerMovement ECS system. It only selects the player entity and will not query any other entities, because player movement logic is different from computer-controlled entities, and we do not want to create side-effects inside systems if it’s not needed.

```
1  [With(typeof(IsPlayerComp), typeof(VelocityComp))]
2  public sealed class PlayerMovementSystem : AEntitySetSystem<float>
3  {
4      protected override void Update(float ∆Time, in Entity entity)
5      {
6          ref var velocity = ref entity.Get<VelocityComp>();
7          var rb = entity.Get<RigidbodyComp>().Value;
8  
9          if (velocity.Direction == Vector2.zero)
10         {
11             rb.velocity = Vector2.zero;
12             return;
13         }
14 
15         rb.velocity = velocity.speed * velocity.Direction;
16     }
17 }
```

Code 5.4: Player movement ECS system

Update function is called in every frame. This system reads values from VelocityComp of a player entity (line 6), and then applies them into the Rigidbody of Unity GameObject which is tied to the player entity (line 15). If inputs are not pressed, a zero vector is sent (lines 9-12).

5.3 Simple enemies

After having implemented a simple player controller, we wanted to create a simple enemy. Goal of a simple enemy is to follow the player character and damage them if the player finds himself too close. Having enemies will be necessary to test out other mechanics implemented in the future, like player shooting projectiles which damage enemies and more mechanics that interact with them.

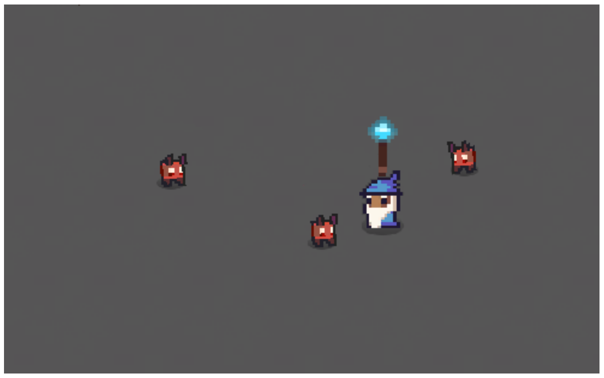


Figure 5.2: Player character entity, and three enemy entities

```
1  [With(typeof(IsEnemy), typeof(VelocityComp))]
2  public sealed class EnemyFollowSystem : CustomEntitySetSystem
3  {
4      public EnemyFollowSystem(World world) : base(world, ...
           SystemParameters.UsePlayer, interval: 0.2f)
5      {
6      }
7
8      protected override void Update(float ∆Time, ReadOnlySpan<Entity> ...
           entities)
9      {
10         var playerPosition = Player.Get<PositionComp>().Value;
11
12         for (int i = 0; i < entities.Length; i++)
13         {
14             ref readonly var entity = ref entities[i];
15             ref readonly var position = ref entity.Get<PositionComp>();
16
17             var normalized = (playerPosition - position.Value).normalized;
18
19             entity.Get<VelocityComp>().Direction = normalized;
20         }
21     }
22 }
```

Code 5.5: Enemy follow ECS system

In line 1, EnemyFollowSystem selects every entity flagged as enemy (meaning entities with an IsEnemy component), that also has a VelocityComp.

Update function inside the system gets called once every frame. We get Player Entity's position (line 10). Then the function iterates over every queried entity and calculates the direction it should go to, to get closer to the Player Entity (lines 12 - 17). That value is then saved in VelocityComp.Direction for each entity (line 19).

5.3.1 Velocity System

To control movement of entities, we created a VelocitySystem ECS system which will take care of most moving entities. The system reads values from the VelocityComp component. This system only contains logic which moves entity to the direction pointed by a 2D vector, by a specified speed. Other systems were created to calculate the direction vector and speed. This is done so we can use VelocitySystem for different types of enemies and entities like projectiles later on, even if they have different behaviors.

VelocitySystem will first select every entity which has a VelocityComp. Then move each of those entities to the direction pointed by the vector and speed, which is read from the VelocityComp.

5.3.2 Enemy Melee Attack System and Attack System

EnemyMeleeAttackSystem queries all entities with the flag component IsMeleeEnemy, and then checks if the enemy is close enough to the player entity to perform an attack.

From now on we will only include code of Update function inside systems and/or relevant functions, which is the hotpath of every ECS system and hold entire logic. Inside, the function iterates over every queried entity, which in this system are entities with IsMeleeEnemy component.

```
protected override void Update(float state, ReadOnlySpan<Entity> entities)
{
    var playerPosition = Player.Get<PositionComp>().Value;
    var damageEvents = Player.Get<DamageEventsComp>().DamageEvents;

    for (int i = 0; i < entities.Length; i++)
    {
        ref readonly var entity = ref entities[i];
        ref var enemyAttack = ref entity.Get<AttackComp>();

        if (!enemyAttack.CanAttack)
            continue;

        if (Vector2.Distance(entity.Get<PositionComp>().Value, ...
            playerPosition) >
                DistanceToHit + entity.Get<ScaleComp>().Magnitude * 1.7f)
            continue;

        enemyAttack.CanAttack = false;

        ref readonly var damage = ref entity.Get<DamageComp>();
        damageEvents.Add(new DamageEvent() { Source = entity, ...
            DamageType = damage.DamageType, Damage = damage.Value });
    }
}
```

Code 5.6: Enemy melee attack ECS system

```
protected override void Update(float state, ReadOnlySpan<Entity> entities)
{
    for (int i = 0; i < entities.Length; i++)
    {
        ref readonly var entity = ref entities[i];
        ref var attack = ref entity.Get<AttackComp>();
        if (attack.CanAttack)
            continue;

        attack.Timer += state;
        if (attack.Timer < attack.Cooldown)
            continue;

        attack.CanAttack = true;
        attack.Timer = 0f;
    }
}
```

Code 5.7: Attack ECS system

Enemy attacks have a cooldown, it's so enemies do not deal damage to the player entity every frame when they're close to him. Cooldown time is stored in the AttackComp component on enemy entities. The AttackSystem ECS system ticks up a timer on every

AttackComponent (5.3.2 line 10) until it is higher than the cooldown in AttackComp. When the timer reaches a certain threshold, it resets itself, and then it changes the bool flag CanAttack inside the component to true (5.3.2 lines 14-15).

When an enemy is close enough to the player entity, and the CanAttack flag has been set to true (5.3.2 lines 11-16), an attack becomes successful in the the EnemyMeleeAttackSystem. Which further makes a new damage event inside a DamageEventsComp component on the player entity, and sets the CanAttack flag to false (5.3.2 lines 18-21). The DamageEventsComp component holds data of all the received attacks, and will be later used by other systems which read those events and reduce health of entities.

The first parameter of the Update function - state - is the time between last and current frame given in seconds. Not every system needs it, but the AttackSystem here uses it to calculate timers.

5.4 Health and Damage Systems

Since we have logic that creates DamageEvents, we need systems that handle these events, which will apply damage numbers into the HealthComp component, and kill entities if they get to 0 health.

5.4.1 Damage System

DamageSystem ECS system is tasked with taking damage events from DamageEventsComp, and applying them onto the health of an entity, essentially taking away their health value when they get damaged. It handles damage done on enemies, and not the player entity. We needed finer damage control between these two entity types, and didn't want to include side-effects into the DamageSystem.

```
protected override void Update(float state, ReadOnlySpan<Entity> entities)
{
    ref var playerHealth = ref Player.Get<HealthComp>();
    ref var score = ref World.Get<ScoreComp>();

    for (int i = 0; i < entities.Length; i++)
    {
        ref readonly var entity = ref entities[i];
        var damageEvents = entity.Get<DamageEventsComp>().DamageEvents;

        if (damageEvents.Count == 0)
            continue;

        if (entity.Has<InvulnerableComp>())
        {
            damageEvents.Clear();
            continue;
        }

        HandleDamageEvents(in entity, damageEvents, ref playerHealth, ...
            ref score);
    }
}
```

Code 5.8: Damage ECS system update function

HandleDamageEvents function holds logic which reduces health value in HealthComp component on an entity (5.4.1 line 9). We decided to split up the Update function into two functions to make the code more readable.

HealthComp has only 2 fields, Current and Max. HealthComp.Currect contains entity's current hp. HealthComp.Max is used as a starting and max point, where the HealthComp.Current value can't be higher than HealthComp.Max. HealthComp.Max is also used by upgrades with interactions like increasing hp.

```
private void HandleDamageEvents(in Entity entity, List<DamageEvent> ...
    damageEvents, ref HealthComp playerHealth)
{
    float totalDamage = 0f;

    for (int i = 0; i < damageEvents.Count; i++)
    {
        var damageEvent = damageEvents[i];

        ref var health = ref entity.Get<HealthComp>();
        float damage = damageEvent.Damage;

        totalDamage += damage;
        health.Current -= damage;
    }

    World.Publish(new DamageDealtEvent(entity, totalDamage));
    damageEvents.Clear();
}
```

Code 5.9: Damage ECS system HandleDamageEvents function

We omitted big parts of the function here, for clarity. The logic inside is more complex, factoring in player upgrades, enemy damage resistances, player multipliers, special player class perks, etc.

5.4.2 Health System

Now we need a system that will go over every entity with a HealthComp component, and check if its HealthComp.Current value is less or equal to 0 (5.4.2 line 6). If it is, the system will set a new DestroyComp component on that (5.4.2 line 8), as it is presumed dead in the game's context.

```
1  protected override void Update(float state, ReadOnlySpan<Entity> entities)
2  {
3      for (int i = 0; i < entities.Length; i++)
4      {
5          ref readonly var entity = ref entities[i];
6          if (entity.Get<HealthComp>().Current ≤ 0)
7          {
8              entity.Set<DestroyComp>();
9          }
10     }
11 }
```

Code 5.10: Update function of health ECS system

DestroySystem destroys every entity with DestroyComp in every frame, it's also used on other entities like projectiles, which disappear after some time in flight, or on enemy hit.

To kill an entity in DestroySystem, the Dispose method of ECS entity is used (5.4.2 line 6).

```
1 protected override void Update(float state, ReadOnlySpan<Entity> entities)
2 {
3     for (int i = 0; i < entities.Length; i++)
4     {
5         ref readonly var entity = ref entities[i];
6         entity.Dispose();
7     }
8 }
```

Code 5.11: Update function of destroy ECS system

5.5 Player projectiles

Since we have both a player character that can move around, and enemies that follow the player and can damage him, we implemented mechanics for the player to interact with enemies. The first thing we implemented is an ability to shoot projectiles which can damage enemies. The player uses the mouse cursor to point a wand in a direction where projectiles will travel. These projectiles are fired automatically without any extra input from the player.

A basic projectile will hit the first enemy on its path, reduce his health by some amount and then it will be disposed of. There are more complicated projectiles for player to get later on, like:

- Area-of-effect projectile which will damage every enemy around in a given radius when it hits.
- Piercing projectile which will not be disposed after hitting the first enemy, but continue damaging enemies in a line.
- Homing projectile which will guide itself onto enemies, so the player does not have to be 100% accurate to score a hit.

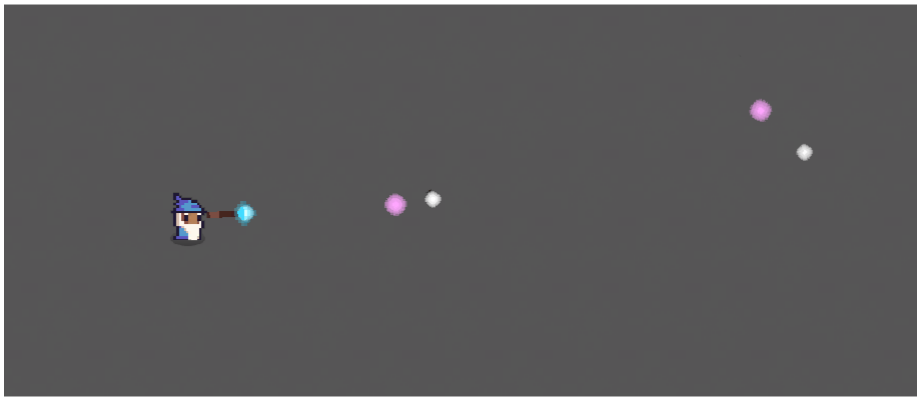


Figure 5.3: Player entity shooting projectile entities

5.6 Arena bounds for enemies

To control how far the player and enemies can go and keep then in a square-shaped arena, we created a bounds area around the playable arena. Bounds also limit where enemies spawn, enemy spawn zones are calculated from bounds area, and the current player position. Making it so that enemies only spawn where the player can't see them. Bounds area is represented by a ground sprite, so a player will be able to see what is the limit of their movement.

5.7 Wave system

To have a basic playable experience we created general control systems that control and send events to other systems, like starting new rounds and spawning enemies to keep the game running. WaveSystem controls all that. We have decided that the a single game round should last about 30 to 60 seconds, during which enemies will be spawned. The player's main goal will be to survive, and then his secondary goal will be to get as much gold as possible to be able to buy upgrades. After the time goes out, the game stops and the shop menu pops up. When the player is done he can exit the shop menu, and then the game time resumes and WaveSystem starts the next round. Rounds consist of diverse sets of enemies, they also progressively get harder so a player has to keep up with the difficulty. He gets further help from the UpgradeSystem to do that.

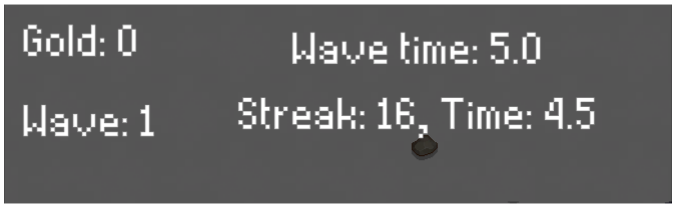


Figure 5.4: Wave UI

5.8 Upgrade system

Roguelike genre is characterized by giving the player a randomized experience in every game. Many games achieve this by giving the player some random effects which changes how the game plays. We wanted to implement this in our game by showing a shop menu for a player after every wave. In there, player will be able to pay for upgrades with gold.

Players performance is being judged on having a high kill streak, avoiding being hit, just to name a few. The higher the numbers, the higher the reward. But there's still a small amount of base gold given out each round.

When the shop menu shows up, there are 3 weighted random upgrades available to buy. A player can buy all 3 upgrades, as long as he has enough gold. If the player does not have enough gold to buy an upgrade he wants, he can "lock" the upgrades. This means that next time the upgrades menu shows up, it will have the exact same 3 upgrades. On the other hand, if the player has enough gold to buy more upgrades, or is not satisfied with what they can choose from, he's able to pay to reroll and get 3 new upgrades to choose from.

Upgrades have a wide range of effects, from the simplest ones increasing player's statistics like damage or speed, to more complex one's like changing the way player projectiles work. Those more complex one will have a smaller chance of appearing in a roll. There's 5 tiers of rarity – grey, green, blue, purple and gold. This is done for easy identifications of how much potential an upgrade can have. Green upgrades cost more than grey ones, and gold cost the most.

Some examples for each upgrade tiers:

- Grey: increased player damage +10%, increased player movement speed +15
- Green: Homing Projectiles +10%
- Purple: Bigger projectile and higher projectile dmg
- Gold: 100% chance for a homing projectile

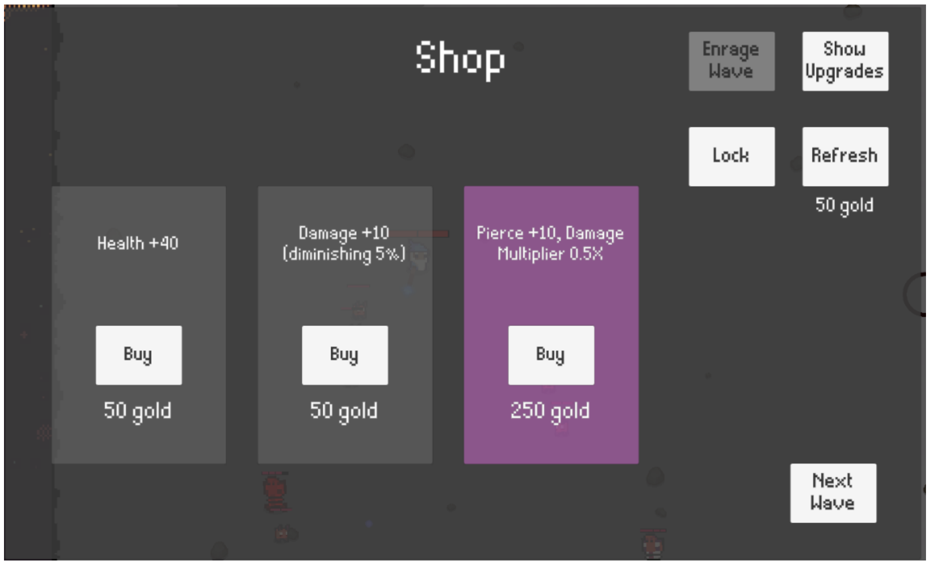


Figure 5.5: Shop menu

5.9 Other features

5.9.1 Game saves

We have implemented a save system, that saves the game state between waves, when the game is on the upgrade's menu. A player can then leave to main menu or exit the game, and the game's state will be saved, and can be loaded later on from the main menu.

Saves data is being saved in different places based on the platform.

Windows: %userprofile%\AppData\LocalLow\UpToSpeedStudios\RoguelikeEcs

Linux: $HOME/.config/unity3d

WebGL: "idbfs/UpToSpeedStudios/RoguelikeEcs/"

The GameStateController is responsible for saving and loading all game state data.

```
using var writer = new JsonTextWriter(textWriter);
writer.Formatting = Formatting.Indented;

writer.WriteStartObject();
writer.WriteValue("gameSaveName", saveName);
writer.WriteValue("gameVersion", DataController.Version.ToString());
writer.WriteValue("gameSaveVersion", gameSaveVersion.ToString());
writer.WriteValue("saveDate", DateTime.UtcNow);

_gameData.Save(writer);

writer.WriteEndObject();

writer.Close();
```

Code 5.12: Saving state core logic

To not have messy classes, with both game logic and save state logic, we decided to make a special directory for save state extensions. The directory being "Source/Assets/Scripts/Core/StateExtensions".

This way, we have all save and load state logic in one place. So there's no need to search after them in the repository.

```
public static void Save(this GameData gameData, JsonTextWriter writer)
{
    writer.WritePropertyName("totalPlayTime");
    writer.WriteValue(gameData.TotalPlayTime);
    writer.WritePropertyName("totalKills");
    writer.WriteValue(gameData.TotalKills);
    ...

    writer.WritePropertyName("unlockedUpgrades");
    writer.WriteStartArray();
    foreach (var type in gameData.UnlockedUpgrades)
        writer.WriteValue(type.ToString());
    writer.WriteEndArray();

    ...
}
```

Code 5.13: Saving state extension method

We decided that the saves should be stored in a human readable format, so we chose JSON.

There's two types of game state saves, one is for the general user profile, the second is for gameplay run states. The general user game data save is loaded every time the player starts the game. Gameplay run states on the other hand, can only be loaded from main menu, to load in the saved run.

```json
{
  "gameSaveName": "game_data",
  "gameVersion": "0.1.2",
  "gameSaveVersion": "0.1.0",
  "saveDate": "2023-05-13T17:15:24.005108Z",
  "totalPlayTime": 0.0,
  "totalKills": 829,
  ...
  "unlockedUpgrades": [
    "AlwaysAoENeverPierce",
    "SmallProjectiles",
    "BigProjectiles",
    "MultipleProjectilesRapidFire"
  ]
  ...
}
```

Code 5.14: Game data save state

```json
{
  "saveName": "save9",
  "gameVersion": "0.1.2",
  "saveVersion": "0.1.0",
  "playTime": 29.8,
  "saveDate": "2023-05-13T17:14:57.472231Z",
  "automaticAim": false,
  "areaSize": "Medium",
  "difficulty": "Normal",
  "waveInfo": {
    "time": 0.0,
    "waveNumber": 1,
    "waveFinished": true
  },
  "shopUpgrades": [
    {
      "upgradeType": "AttackSpeed",
      "isBought": false
    }
  ],
  "score": {
    "kills": 0,
    "bossesKilled": 0,
    "damageDealt": 0.0
    ...
  },
  "gold": 111.2985,
  "characterType": "Default",
  "playerPosition": {
    "x": 253.26,
    "y": 192.26
  },
  "upgrades": {},
  "enemies": [
    {
      "type": "Basic",
      "position": {
        "x": 183.69,
        "y": 231.22
      },
      "health": 0.2
    },
    {
      "type": "Basic",
      "position": {
        "x": 215.85,
        "y": 230.46
      }
    }
  ]
}
```

Code 5.15: Gameplay run save state

5.9.2 Enemies changing visuals as health points indicator

Initially we wanted to avoid extra entities/UI like health bars. So instead we implemented a similar mechanic that does the same job of displaying entities health percentage. Without creating new entities or UI, by first changing the saturation of the enemy sprites, and then later changing brightness instead.

```
1  private const float MinBrightness = 0.35f;
2
3  protected override void Update(float state, ReadOnlySpan<Entity> entities)
4  {
5      for (int i = 0; i < entities.Length; i++)
6      {
7          ref readonly var entity = ref entities[i];
8          ref readonly var health = ref entity.Get<HealthComp>();
9          ref var renderer = ref entity.Get<RendererComp>();
10
11         if (health.IsMax)
12         {
13             if (!renderer.HasOriginalColor)
14             {
15                 renderer.Value.color = renderer.OriginalColor;
16                 renderer.HasOriginalColor = true;
17             }
18
19             continue;
20         }
21
22         Color.RGBToHSV(renderer.OriginalColor, out float hue, out float ...
               saturation, out float brightness);
23         brightness = Mathf.Lerp(MinBrightness, brightness, health.Percent);
24         renderer.Value.color = Color.HSVToRGB(hue, saturation, brightness);
25
26         renderer.HasOriginalColor = false;
27     }
28 }
```

Code 5.16: Enemy health rendering ECS system

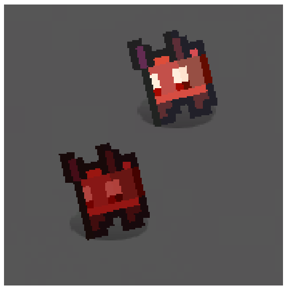


Figure 5.6: Healthy (top right) and damaged (bottom left) enemies

If the entities health is max, we skip it (5.9.2 lines 11-20). Otherwise we calculate the entities sprite brightness, lerp it between minimum and max brightness. Then set it back by calculating it back to RGB (5.9.2 lines 22-24).

This way, entities with low health will appear as "watered down", and unhealthy.

5.9.3 Health points bar

The player entity got a health bar early on, an UI element that shows the current status of player's health values. But none of the enemies had one.

We decided that displaying health as brightness wasn't enough information for the player. So we figured that we should make a health bar for boss enemies. To better articulate the damage done to them, since they have much higher health values than normal enemies.

```
public struct HealthBarComp
{
    public RectTransform Transform;
    public Image HealthBarFgImage;
    public Image HealthBarDamageTakenImage;

    public float LastHealthPercent;
    public float DamageTakenTimer;
    public float LastMaxHealth;
}

protected override void Update(float state, in Entity entity)
{
    ref var healthBar = ref entity.Get<HealthBarComp>();

    ref readonly var health = ref entity.Get<HealthComp>();
    float healthPercent = health.Percent;

    healthBar.DamageTakenTimer += state;

    if (Math.Abs(healthBar.LastHealthPercent - healthPercent) < 0.001f
        && healthBar.LastMaxHealth == health.Max)
        return;

    healthBar.HealthBarFgImage.fillAmount = healthPercent;
    healthBar.LastHealthPercent = healthPercent;
    healthBar.LastMaxHealth = health.Max;

    if (healthBar.DamageTakenTimer < 0.3f)
        return;

    healthBar.HealthBarDamageTakenImage
        .TweenImageFillAmount(healthPercent, 0.5f);
    healthBar.DamageTakenTimer = 0f;
}
```

Code 5.17: Health bar rendering ECS system

For health bars we used a known UI and game dev concept, known as tweening. Tween is a word mostly used in software UI contexts, it's an animation term that means "in between".

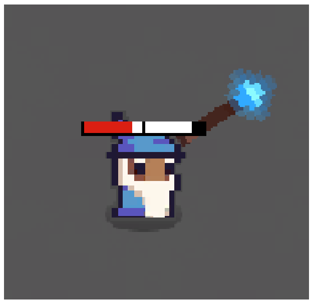


Figure 5.7: Player entity taking a heavy hit

It's used for gradually changing some kind of value, it can be height, size, transparency, anything. Giving the object a dynamic feeling of smooth movement.

We have three health images, red foreground, black background, and white middle ground. Red foreground fill amount is changed right away with health percentage changes. Black background fill amount is never changed. While the white background fill amount is changed with tweening to the current health percent, basically meaning that the image fill amount will interpolate gradually with time until it gets to it's destination.

This feature was implemented to give players health more dynamic interactivity.

5.9.4 Circle effect

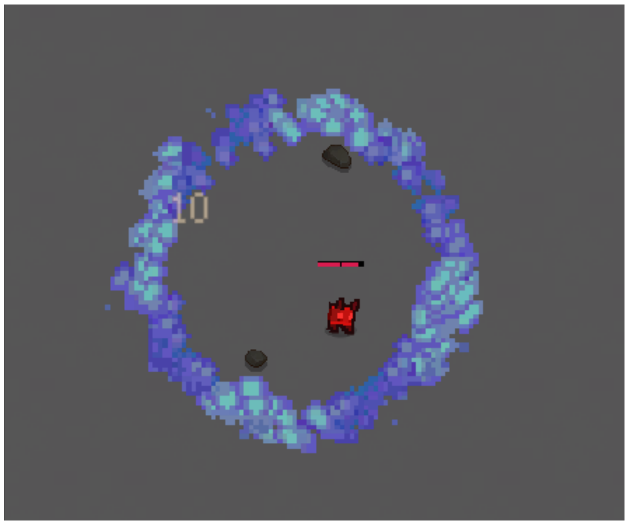


Figure 5.8: Visuals of the circle effect

When testing, we saw that the game was lacking a positional objective. The player was running around and only dodging enemies. So we created an incentive to move to a certain spot on the map.

When player gets to the position, an area-of-effect damage is dealt to enemies in certain radius, in form of few circles expanding from the original position.

```
 1  protected override void Update(float state, in Entity entity)
 2  {
 3      ref var circleEffect = ref entity.Get<CircleEffectComp>();
 4      circleEffect.Timer += state;
 5
 6      if (circleEffect.Timer ≥ circleEffect.Duration)
 7      {
 8          entity.Set<DestroyComp>();
 9          return;
10      }
11
12      circleEffect.EffectTimer += state;
13      if (circleEffect.EffectTimer < circleEffect.EffectInterval)
14          return;
15
16      circleEffect.EffectTimer = 0;
17
18      ref readonly var position = ref entity.Get<PositionComp>();
19      ref readonly var radius = ref entity.Get<RadiusComp>();
20      ref readonly var damage = ref entity.Get<DamageComp>();
21
22      int hits = _physicsScene.OverlapCircle(position.Value, radius.Value ...
            * _aoeRadiusMultiplier, _contactFilter, _colliders);
23
24      World.Publish(new AoEAnimationEvent(AnimationType.Circle, ...
            position.Value, radius.Value));
25
26      for (int i = 0; i < hits; i++)
27      {
28          ref readonly var hitEntity = ref ...
                _colliders[i].GetComponent<EnemyReference>().Entity;
29
30          hitEntity.Get<DamageEventsComp>().DamageEvents.Add(new ...
                DamageEvent()
31              { Source = entity, Damage = damage.Value, DamageType = ...
                    damage.DamageType });
32      }
33  }
```

Code 5.18: Circle effect system update function

In 5.9.4 lines 3-14 calculate how long the effect stays up and time between hits. In line 22 we calculate all area of effect hits in a radius. Line 24 in starts the circle effect animation.

Lines 26-31 inapply damage to enemies hit by the circle effect.

5.9.5 Dev UI menu

To make our lives easier, we made a in-editor dev menu. It was mostly used for very general commands/scenarios, for example: disabling certain systems, like WaveSystem, giving the player gold, killing all enemies on the map, saving and loading save states, finishing/starting a new round, and giving the player upgrades.

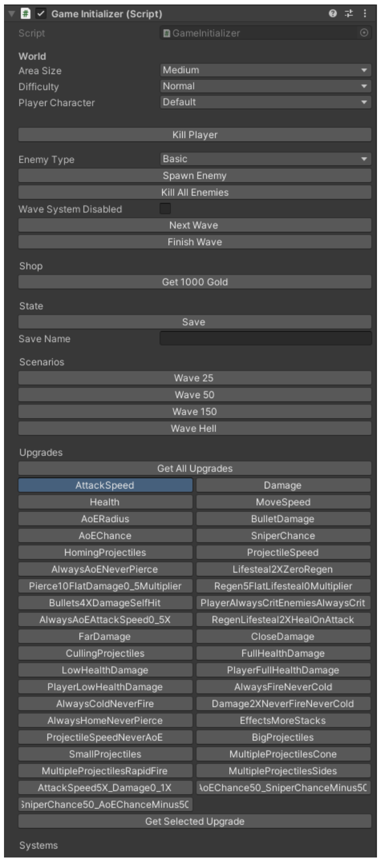


Figure 5.9: Custom in-editor UI dev menu

The dev menu was the best quality of life feature we made for ourselves. It helped us testing certain scenarios very fast, by being able to load the state in and out with a single button click. This way, we could identify and debug bugs much faster, compared to a manual setup.

5.10 More complicated enemies

To create variety of content for our game we wanted more enemies, so the player will have to change how he approaches the game slightly, and can't abuse a one size fits all strategy. This will be done by altering enemies' behavior and their interactions with projectiles, the player and other enemies.

We have made a total of 20 enemy types. We do not want to take too much time presenting each one, so we have chosen 2 types which we think are interesting.

5.10.1 Splitter enemy

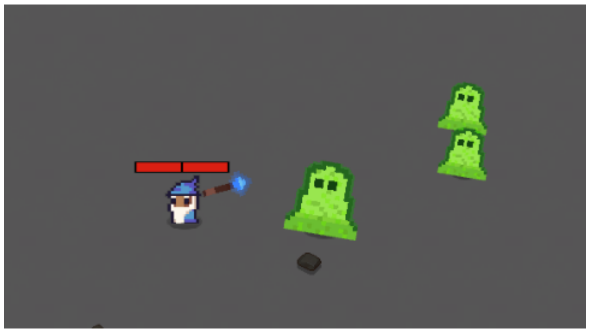


Figure 5.10: Splitter enemy

The splitter enemy follow the player and damages him if he gets to close. What is different about him is that after being defeated, he splits into few smaller versions of himself. It has the capability to split itself how many times we want, each time lowering its damage, health and all the other stats by half. But that would make the enemy unbalanced, so we decided it should split once into two entities, and the boss version should split twice into three entities.

5.10.2 Buffer enemy

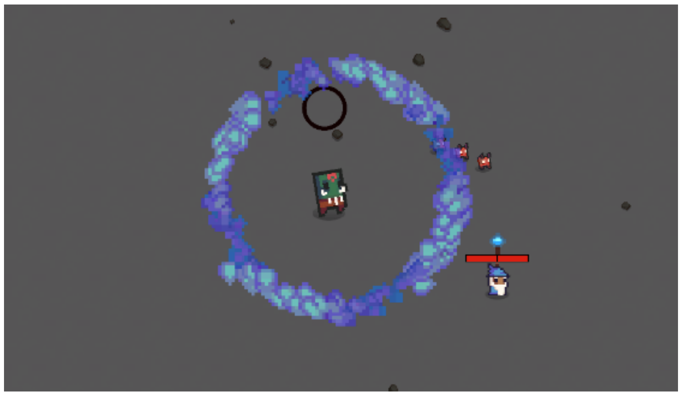


Figure 5.11: Buffer enemy

The buffer enemy tries to keep his distance from a player, and cannot damage him. What it does is regenerate health of enemies nearby to him, and buffs them by increasing their speed and damage. This effect doesn't stack with other buffer enemy effects, for balance reasons.

5.11 Main menu

We didn't want the first round to instantly begin when our game is launched. So we made a starting menu which will be loaded first. The player will be able to customize game's settings, and other things like player character classes and game's diffuculty in there before starting the gameplay.

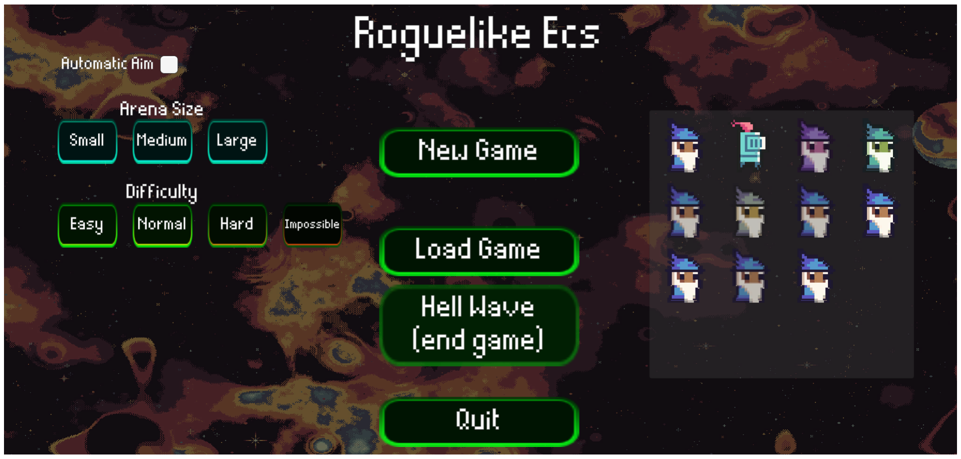


Figure 5.12: Main menu UI

5.12 Game visuals

5.12.1 Outline shader

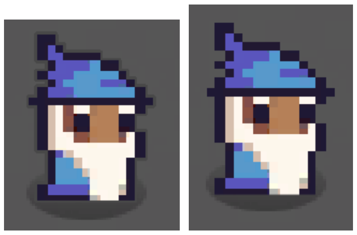


Figure 5.13: Player entity with (left) and without (right) an outline shader

We wanted for the player to stand out even more, from enemies and projectiles. The color pallette, health bar and central position weren't enough. So we decided to use a minor shader that creates an outline around the sprite. The effect is very minimal, but that's by design. The outline is one pixel in length.

5.12.2 Shadows

To make our game feels less flat, we used a simple shadow textures below every entity. That scales with entities size.

```
var shadow = new GameObject("Shadow");
shadow.transform.parent = go.transform;
shadow.transform.localPosition = new Vector3(0, -0.8f, 0);
var shadowRenderer = shadow.AddComponent<SpriteRenderer>();
shadowRenderer.sprite = Resources.Load<Sprite>("Textures/Shadow");
entity.Set(new ShadowRendererComp { Value = shadowRenderer });
```

Code 5.19: Shadow gameobject creation

```
 1  private float _slowedTimer;
 2
 3  protected override void Update(float state, ReadOnlySpan<Entity> entities)
 4  {
 5      _slowedTimer += state * 0.1f;
 6
 7      for (int i = 0; i < entities.Length; i++)
 8      {
 9          ref readonly var entity = ref entities[i];
10
11          var transform = entity.Get<ShadowRendererComp>().Value.transform;
12          float ping = Mathf.PingPong(_slowedTimer, 0.05f);
13
14          transform.localScale = Vector3.one * (1 + ping);
15          transform.localPosition = new Vector3(-0.025f + ping, -0.775f + ...
                ping, 0);
16      }
17  }
```

Code 5.20: Shadow render animation ECS system

The ShadowRenderingSystem is responsible for giving the shadows apparent animations. The shadows move around, and scale a tiny bit every frame, making them feel dynamic and responsive to entities movement. Furthermore making an illusion of actual animations.

In 5.12.2 12 we calculate a ping pong value between two values, which means we have a value which goes from one number to the other with a given input, here being a slowed timer. We use that value in line 14 to change scale of a shadow up and down, so that it appears to be moving slightly. Line 15 moves the shadow along the entities.

5.12.3 Random rocks

To make the background more interesting, and especially to help the player with spatial orientation, we added random spawning rocks to the game. These are for visuals only, and have no mechanical game impacts.

```
var rocks = Resources.LoadAll<Sprite>("Textures/Rocks/");
var rockParent = new GameObject("Rocks");
for (int i = 0; i < 75 * areaSizeType.GetMultiplier(); i++)
{
    var rock = new GameObject("Rock");
    rock.transform.parent = rockParent.transform;
    var rockRenderer = rock.AddComponent<SpriteRenderer>();
    rockRenderer.sortingOrder = -2;
    var rockSprite = rocks[Random.Range(0, rocks.Length)];
    rockRenderer.sprite = rockSprite;
    rockRenderer.color = new Color(0.4f, 0.4f, 0.4f);
    rockRenderer.flipX = Random.Range(0, 2) == 0;
    rock.transform.position = new Vector3(Random.Range(-areaSize.x, ...
        areaSize.x), Random.Range(-areaSize.y, areaSize.y), 0);
    float scale = Random.Range(1f, 2f);
    rock.transform.localScale = new Vector3(scale, scale, 1);

    var shadow = new GameObject("Shadow");
    shadow.transform.parent = rock.transform;
    shadow.transform.localPosition = new Vector3(0, -0.95f, 0);
    shadow.transform.localScale = new Vector3(1f, 1f, 1f);
    var shadowRenderer = shadow.AddComponent<SpriteRenderer>();
    shadowRenderer.sortingOrder = -3;
    shadowRenderer.sprite = rockSprite;
    shadowRenderer.color = new Color(0f, 0f, 0f, 0.3f);
    shadowRenderer.flipX = rockRenderer.flipX;
}
```

Code 5.21: Randomly generated map rock creation

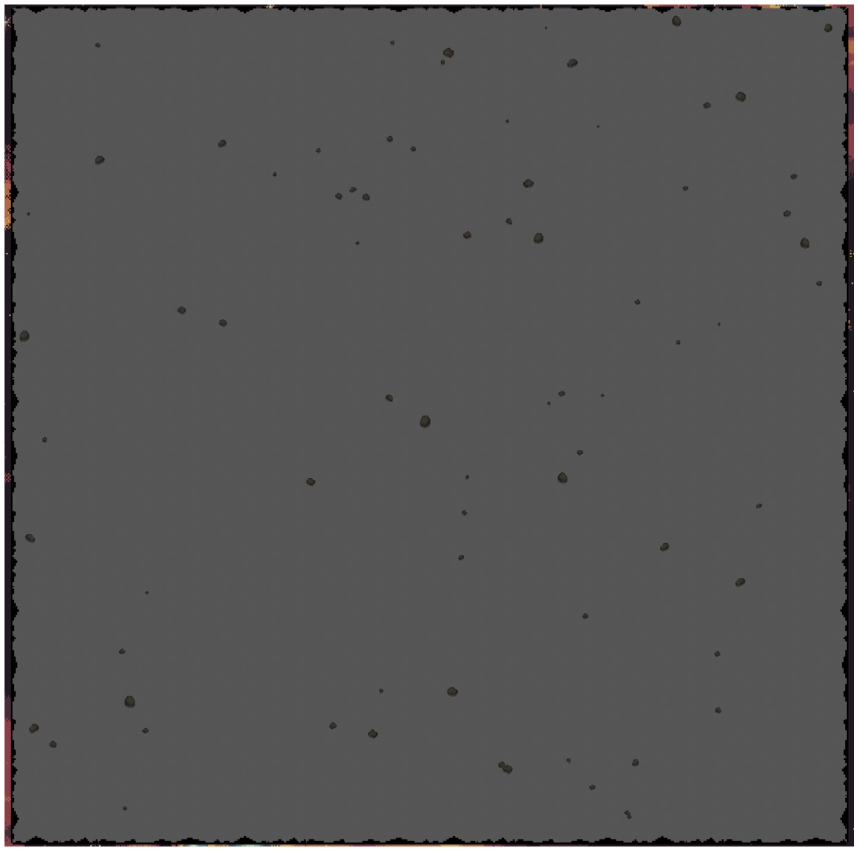


Figure 5.14: Randomly generated rocks on a medium map

We spawn from 52 to 112 rocks, dependent on the map size. The rocks are scaled, flipped and chosen at random.

Rocks also have a shadow that corresponds to the chosen rock sprite, to give them a 2.5D/3D feeling.

5.13 Balancing

To tweak the numbers like health, damage and upgrade's value we needed to playtest the game extensively to find out which of those have to be changed in order to create game as enjoyable as possible.

We did not want the game to feel too easy or too hard, so this was done carefully to find good fitting values.

A difficulty select may help players to tweak the game to their liking, but it also means more balancing for us to do, to do meaningful changes between difficulties.

Chapter 6

Developing games in ECS

6.1 Initial Process

Initially we planned to split the thesis into few time periods:

- 16-29 Jan. ECS Integration
- 1-19 Feb. Prototyping
- 20 Feb - 26 Mar. Core
- 27 Mar - 16 Apr. Polishing
- 17 Apr - 14 May. Documentation

We thought that the first 4 weeks will be used on learning ECS and making a prototype that will be entirely scratched after the prototyping period, this turned out to not be the case.

Learning ECS took much less time than we thought. And the prototype turned out to be of higher quality than expected. Which meant that we were finished with the base product in the start of February. And the game was fully playable in the middle of February. With save states, progression, and a majority of mechanics implemented.

6.1.1 Project Management

At first, we tried using Kanban boards, and scrum sprints. But because we always worked together on the project Monday to Friday every morning. We noticed that extra time spent on managing the boards wasn't worth it. So instead we kept two separate local notepads, with current tasks and thoughts, and shared them with each other each day. Organizing what to prioritize, what needs to be discussed, etc, every morning for 5-15 minutes. This way, we didn't use any fancy systems, but still had a backlog and very fast iteration.

6.2 Common ECS mistakes

6.2.1 Changing struct data by value

In vast majority of C# applications, classes are the common type to use. Classes are reference types, as opposed to structs being value types. By default classes are pointing towards an object in memory, while structs are the object.

With C# 7.2 structs can be passed by reference instead [13], making it so we don't copy the struct each time we pass it. This makes developing with structs much easier and more performant in ECS frameworks.

The biggest hurdle is to learn to automatically use ref keyword every time we change state. And also ideally use ref readonly keywords when we don't need to mutate state.

Here we have the AttackSystem, but with the ref keyword commented out. Making it so we copy the struct state to "attack", and change the values of the copied struct instead of the original. The easiest solution is to use the ref keyword, but one can also go back to the original struct and change it's values directly.

```
protected override void Update(float state, ReadOnlySpan<Entity> entities)
{
    for (int i = 0; i < entities.Length; i++)
    {
        ref readonly var entity = ref entities[i];
        /*ref */var attack = /*ref */entity.Get<AttackComp>();
        if (attack.CanAttack)
            continue;

        attack.Timer += state;
        if (attack.Timer < attack.Cooldown)
            continue;

        attack.CanAttack = true;
        attack.Timer = 0f;
    }
}
```

Code 6.1: Attack ECS system with a ref bug

This was the most common issue that caused bugs when learning ECS.

6.2.2 Return and Continue

The second most common bug creating issue is using the return keyword instead of the continue keyword in loops. This is another common development habit, that needs to be re-learned. It happens because of how OOP works. In OOP most things are objects, which means that when we mutate state, we can use return to safely exit the logic. Because we're mutating state on a single object. This is not the case in ECS and SoA.

This habit causes the for loops to exit prematurely, causing bugs.

6.3 Game Runtime

Most software have a core initiating point, where the developer has full overview over the order of initialization and possibly the update cycle of systems. This is the case in ECS with systems, since they're all stored in one spot, and executed in a specific order (unless ran in parallel).

The DataController class loads data of our program, that is information that will be used inside ECS systems, for example WaveData which stores information about which Enemy entities will be present in each wave, or UpgradeData which holds information about upgrades which player can get. This gets loaded into the memory. The GameController class then initializes ECS world which holds ECS entities, and sets up its parameters like maximum number of entities. It also initializes key features like GameUpdateSystem or UI. GameUpdateSystem update call is made inside GameController for each frame. Core is the main class that holds GameUpdateSystem, which contains all ECS systems in our game which are then refreshed.

All systems are ran sequentially, and the order matters for most systems. For example, DestroySystem is one of the last systems to run. Thanks to that, the game can calculate in systems that are placed before DestroySystem which entities have 0 remaining health points. Then those entities are being removed in the same frame when DestroySystem runs. This is important because ECS usually does not save state between cpu cycles (unless specifically designed to do so) and we could loose a call for entity to do some action.

We could have split systems into better groups, for clarity. But we didn't think it was complex enough to warrant it.

Note that all the systems and groups shown here are executed sequentially.

The first group is the input/player group, it's the most important one and should be first.

```
new InputSystem(world),
new MousePositionSystem(world, camera),
new MouseInputSystem(world),
new PlayerMovementSystem(world, bounds),
new AutomaticTargetingSystem(world),
new WandRotationSystem(world),
```

Code 6.2: Input and player ECS systems

The second group is the enemy spawning/initializing/updating group, it's responsible for creating enemies, and then setting up all the initial values. After that, it handles general enemy mechanics, like following the player, growing, buffing.

```
new WaveSystem(world, dataController.WaveData, dataController.GameData),
new EnemySpawnSystem(world, dataController.EnemyRecipes),
new EnemyFollowSystem(world),
new EnemyFollowMapEdgeSystem(world),
new EnemyFollowDistanceSystem(world),
new SetInitialDestinationSystem(world, bounds),
new EnemySetDestinationSystem(world, bounds),
new EnemyChargeTimeSystem(world),
new EnemyChargeFollowSystem(world),
new EnemyGrowSystem(world), //No dependencies
new EnemyBuffSystem(world),
new BufferEnemySystem(world),
```

Code 6.3: Enemy-centric ECS systems

The next group is about rendering and flashing effects. Flashing effects need to be after the HealthDisplaySystem, so they take priority over manipulating sprites.

```
new HealthDisplaySystem(world),
new FlashEffectSystem(world),
new EnemyAttackedFlashSystem(world),
```

Code 6.4: Rendering effects ECS systems

The next group is responsible for attack and projectile systems.

```
new AttackSystem(world),
new ProjectileSpawnSystem(world),
new ProjectileHomingSystem(world),
new EnemyMeleeAttackSystem(world),
new EnemyRangedAttackSystem(world),
new EnemyRangedDelayedAttackSystem(world),
```

Code 6.5: Attack and projectile ECS systems

This group is a very mixed bag, and they mostly don't have any relations to each other. It handles entity effects, bounds, and velocity. These systems need for the most part just be between the initial groups, and closing systems.

```
new InitColdSystem(world),
new ColdSystem(world),
new PoisonSystem(world),
new BleedingSystem(world),
new CreateCircleEffectSystem(world),
new CircleEffectSystem(world),
new PlayerCurseSystem(world),
new InitEnemyPinballBounds(world),
new EnemyPinballBounds(world, bounds),
new EnemyBoundsCollisionSystem(world, bounds), //Before velocity
new VelocitySystem(world),
new EnemyChargeBoundsCollisionSystem(world, bounds),
//
new RuptureSystem(world),
new ProjectileCollisionSystem(world),
new BurningSystem(world),
new ProjectileBoundsSystem(world, bounds),
new TimeOutSystem(world),
```

Code 6.6: Miscellaneous ECS systems

The rendering group, is all about updating sprites position, scale, and rotation.

```
new RenderingSystem(world),
new EnemyRenderingSystem(world),
new PlayerDirectionRenderingSystem(world),
new ShadowRenderingSystem(world),
```

Code 6.7: Rendering ECS systems

The invulnerability group, it's responsible for handling invulnerable state on enemies and the player. Making it so certain enemies can't be damaged every so often, or that the player can't be damaged while the game was unpaused from the shop menu.

```
new PhaseSystem(world),
new InvulnerableSystem(world),
new UnHittableSystem(world),
```

Code 6.8: Rendering ECS systems

The first closing group, it's where most calculations are finalized and where entities are finally destroyed. Most of these systems need to be close to the very end, because they depend on state changes from previous systems.

```
new HealthRegenSystem(world), //Before DamageSystem
new DamageSystem(world),
new SleepSystem(world),
new PiercingSystem(world),
new PlayerDamageSystem(world),
new CurseSystem(world),
new HealthSystem(world, dataController.GameData),
new HealthBarSystem(world, camera),
new DeathMarkSystem(world),
new ExplodeOnDeathSystem(world),
new SplitSystem(world), //After all AddDestroy systems
new DestroySystem(world),
new KillStreakSystem(world), //No dependencies
```

Code 6.9: Damage, health, destroy ECS systems

The last closing group, is a cleanup group. It resets states for the next frame.

```
new CameraSystem(world, camera, bounds),
new SpawnAreaSystem(world, camera, bounds),
//
new MultiplierSystem(world),
new ResetInputSystem(world),
//
new FrameStateEndSystem(world, dataController.GameData)
```

Code 6.10: Closing cleanup ECS systems

6.4 Creating new systems

The first most important decision when making a new system is to if even make one in the first place. Sometimes it's better to expand on an already existing system, or rarely potentially create a small side effect inside it.

However in most cases, it's better to go with a new system, since it enables for better modularity and cohesion. It's also important to try to make the logic as generic as possible, without including side-effects. This way, many different entities with a few of the same components can use the same logic without any issues. And we don't have to create new systems or side-effects for specific scenarios.

Note, that this can't always be the case. Sometimes the same components will need different logic for entities. Ex. player and enemy movement management.

When we decided that the logic needs a new system, it's time to decide on which components to include, and exclude from the filter/query. Ex. in our project, we have a follow system for the enemies, where they follow the player. Most enemies use this mechanic, but not all. Charging-, pinball-, distance follow-enemies and enemies that move around the maps edge do not follow the player directly. While stationary and sleeping enemies don't follow the player at all. Because of this we need to filter out all these types of enemies. So the include filter becomes:

```
[With(typeof(IsEnemy), typeof(VelocityComp))]
```

Code 6.11: Default moving enemies include filter

And exclude:

```
[Without(typeof(IsChargeEnemy), typeof(DestinationPositionComp), ...
    typeof(IsSleeping), typeof(IsMapEdgeEnemy),
     typeof(IsDistanceFollowEnemy), typeof(IsPinballEnemy))]
```

Code 6.12: Default moving enemies exclude filter

Essentially filtering out all non enemies, and excluding all non-classic follow mechanic enemies.

The execution order of the system can also be important, because the entity in question might be destroyed (disposed/freed), if we put the new system after the destroy system. Other systems that interact with the states, by either reading it, or mutating it, also matter. And need to be considered when placing the system.

Sometimes systems have no dependencies or communication with other systems, ex. ‘KillStreakSystem‘, these are rare.

Another important factor is update rate. How often should the system be updated? There's a few choices.

The safest bet, is always every tick/frame/update cycle. But it's also the least performative one.

The second option is every fixed tick, this is usually between 30 and 60 ticks per second. And is mainly used in physics simulations. Because the tick is always a fixed number, the physics shouldn't change, and in turn it will be deterministic.

The third and last option is on an interval, this is the most performative option, but also the most "dangerous". If one's not careful, it can easily be misused. A terrible system to use interval updating on would be VelocitySystem. Updating the velocity only every 0.1

seconds would result in choppy and teary visuals. Collision logic and physics would also be affected. That being said, interval update does have a lot of uses. State that needs to be updated rarely, or checked every few seconds is suits it perfectly. EnemyFollowSystem uses an interval of 0.2 seconds, it's to make the following enemies feel more organic and give them a "human reaction" time direction change. It also saves a bit on performance when we have a lot of enemy entities.

6.5 Access to common/shared states/entities

The main entity of the project was the player entity. Around 18 systems need the player's data. But most of these system don't want to work on the player entity, but only want access to it, and read its state.

We decided to create a custom system called "CustomEntitySetSystem" that supplies our player entity if desired. This way, we can access the "Player" entity with a "Player" field in the system. Skipping making filters/queries each time.

The other common shared states are all on the world. And the world is always stored in every system.

```
world.SetSingletonInstance(ScoreComp.Default);
world.SetSingletonInstance<GoldComp>();
world.SetSingletonInstance<KillStreakComp>();
world.SetSingletonInstance(UpgradesComp.Default);
world.SetSingletonInstance(EnemyUpgradesComp.Default);
world.SetSingletonInstance(new SpawnAreaComp { Value = new ...
    SpawnArea(bounds) });
world.SetSingletonInstance<WaveInfoComp>();
world.SetSingletonInstance(new DeadEnemiesListComp { Value = new ...
    List<DeadEnemiesListComp.DeadEnemyInfo>() });
world.SetSingletonInstance(new AreaSizeComp { Value = areaSizeType });
world.SetSingletonInstance(new DifficultyComp { Value = difficulty });
world.SetSingletonInstance(new ShopUpgradesComp { Upgrades = new ...
    UpgradeType[3], IsBought = new bool[3] });
```

Code 6.13: Singleton world components

If the ECS framework didn't support having components on the world. We'd need to make a new singleton entity called "GameState" that can be accessed by any system like "Player" entity.

6.6 Non-ECS Systems

Not every system should be an ECS system, ECS systems have two area with design problems. First one being UI, second one being hierarchy.

The UI was entirely coded in EC and Unity GameObjects, for a few reasons. The main one was that it's much harder to create and manage UI with ECS. UI as a software concept also isn't suited well for ECS, because the behaviors are mostly identical. And most of the UI for the most part doesn't need to be updated very often. So it's usually a better choice to make UI outside of the ECS context.

Hierarchy is harder to achieve in ECS, because of how ECS works. Some hacks and features have been introduced to make this concept a thing in ECS, but they're not the cleanest. The simplest one, is to have a components for child and parent connections. And go through them each time we want do some hierarchical action. The second newer way, more integrated with ECS one is entity relations. It works just how components flags work, but instead of being a single generic flag, it's a component that acts as a verb for another entity. An example of this would be: "ParentEntity" is "ParentComponentRelation" of "ChildEntity" [21]. That being said, DefaultECS doesn't have entity relations, and we decided that it's better to have both UI and hierarchal objects outside of ECS.

Systems that don't need to be updated, usually shouldn't be ECS systems. Both "Data-Controller" and "GameController" are responsible for only initiating the data and game state respectively. More examples on non-ECS systems are: GameStateController, PauseC-ontroller, SceneController, and UpgradeController. All of these aren't updated in intervals or ticks, but instead by messages/events. They still can access ECS entities, through the world object. But aren't a part of the core execution loop.

Chapter 7

Design

7.1 Technical challenges

7.1.1 Unit Testing Games

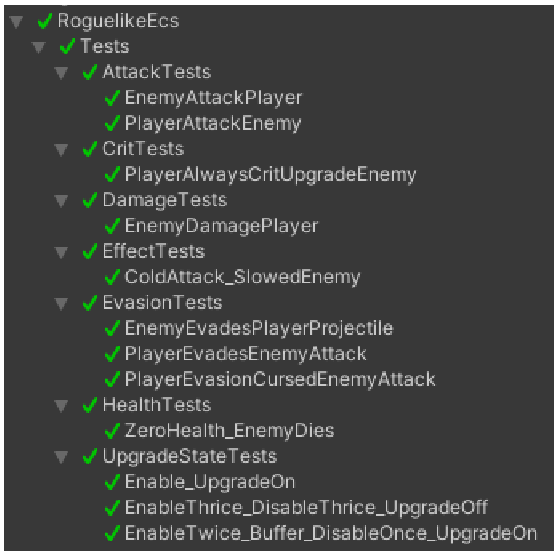
Unit testing games in general is a much harder task than unit testing other types of software.

One of the main reasons is that logic is often mutated over the game's lifetime. In a web application, the client code will be made once, then tweaked a bunch of times, but without almost ever changing the core logic.

This is not the case in games. Mechanics get changed and balanced all the time, often even scrapped for something entirely new.

That being said, we still did some basic unit testing of logic that we knew for a fact won't change.

Even this basic set of unit tests helped a lot. It told us right away if we introduced an exception in the general runtime. They don't cover the majority of niche errors. But making unit tests for these in games where the developer doesn't have full control over everything is usually a bad use of time.



```
▼ ✓ RoguelikeEcs
  ▼ ✓ Tests
    ▼ ✓ AttackTests
        ✓ EnemyAttackPlayer
        ✓ PlayerAttackEnemy
    ▼ ✓ CritTests
        ✓ PlayerAlwaysCritUpgradeEnemy
    ▼ ✓ DamageTests
        ✓ EnemyDamagePlayer
    ▼ ✓ EffectTests
        ✓ ColdAttack_SlowedEnemy
    ▼ ✓ EvasionTests
        ✓ EnemyEvadesPlayerProjectile
        ✓ PlayerEvadesEnemyAttack
        ✓ PlayerEvasionCursedEnemyAttack
    ▼ ✓ HealthTests
        ✓ ZeroHealth_EnemyDies
    ▼ ✓ UpgradeStateTests
        ✓ Enable_UpgradeOn
        ✓ EnableThrice_DisableThrice_UpgradeOff
        ✓ EnableTwice_Buffer_DisableOnce_UpgradeOn
```

Figure 7.1: Unit tests in editor

7.1.2 Unit Testing ECS

Unit testing ECS is a bit different from unit testing non-ECS code. This is because in non-ECS code the data and logic are tied together. So if we call a function on the player, to move it five units to the right, it will do so right away.

And since the data and logic are separated in ECS. We need to call update on the systems, to make the data change. This can also lead to problems when updating other systems. Because it can produce side effects, that break the test. Example if the game has boundaries, and trying to move the player towards the right will make it hit the boundary. Blocking the player. And the test will fail. One solution to this is only enabling/running the systems that are needed for the test.

We opted to not create unit tests with the Unity physics engine, because of a few reasons:

- Unit tests became very slow, average normal test took around 4ms. With physics and scene loading it took 5 seconds. This was unacceptable.
- We didn't need or want to test the physics mechanics in each unit test.

A simple example of a unit test that avoids using physics:

```
[Test]
public void PlayerAttackEnemy()
{
    //Don't destroy projectile on hit and enemy on death
    DisableSystems(SystemType.Destroy, SystemType.Health);

    //Simulated projectile spawn
    var projectile = ProjectileSpawnSystem.SpawnProjectile(in ...
        Player.Get<PositionComp>().Value,
        in Player.Get<PlayerMouseClickComp>().Position,
        ref Player.Get<AttackDataComp>());

    //Simulated collision
    ProjectileCollisionSystem.HitUnit(in projectile, in Enemy);
    UpdateSystems();

    float damage = projectile.Get<DamageComp>().Value;
    Assert.AreEqual(EnemyHealth - damage, Enemy.Get<HealthComp>().Current);
}
```

Code 7.1: Player entity attacking enemy entity unit test

It isn't as simple as a non-ecs test, but it's much simpler than a test with engine physics.

As we can see, testing ECS has a few caveats and quirks. The first one being that all the systems are enabled and ran by default. We could split up the core systems into groups, or only make the systems we need to use, but this isn't an ideal approach. Because it might introduce bugs later on when we add new systems that work with currently unit tested systems.

That's why we chose to disable the systems that would ruin the test instead, by introducing not wanted side effects, of a certain scenario. In the test above we disable "Destroy" and "Health" systems, which are responsible for destroying entities. This way we can check the value amounts and conditions on both tested entities.

Another example of a simpler test. Here we disable the wave system, so it doesn't spawn more new enemies, and ruin the test environment. We also disable health system, so the player can't die and game over.

Then we set evasion to 100%, and try to attack the player 100 times. Each attack should miss.

```
[Test]
public void PlayerEvadesEnemyAttack()
{
    DisableSystems(SystemType.Health, SystemType.Wave);

    Player.Get<EvasionComp>().Chance = 1f;

    for (int i = 0; i < 100; i++)
        UpdateSystems(Enemy.Get<AttackComp>().Cooldown);

    Assert.AreEqual(PlayerHealth, Player.Get<HealthComp>().Current);
}
```

Code 7.2: Player entity evading an enemy entity attack unit test

7.1.3 Unit testing physics with a game engine and ECS

In general unit testing games is harder, compared to unit testing non-game software. It's especially hard because of the lack of direct control over the physics engine and not having full control over object lifetimes. Game developers can often introduce non-deterministic behavior, which further complicates matters.

Ex. This is a unit test to check if the enemy goes towards the player with unity physics. Because we don't have direct access to object lifetimes, and update physics cycles, we need to use "yield return null" game runtime test. This complicates the test a lot. Not only do we have to update our ECS systems, but also the internal physics ticks, this introduces some friction.

```
[UnityTest]
public IEnumerator EnemyMoveToPlayer()
{
    Player.Get<HealthComp>().Current = int.MaxValue;
    Enemy.Get<HealthComp>().Current = int.MaxValue;
    var startPosition = new Vector2(20, 0);
    Enemy.Get<PositionComp>().Value = startPosition;
    Enemy.Get<GameObjectComp>().Value.transform.position = startPosition;

    Assert.AreNotEqual(Player.Get<PositionComp>().Value, ...
        Enemy.Get<PositionComp>().Value);

    UpdateSystems(DeltaTime);
    yield return null;

    Assert.AreNotEqual(startPosition, Enemy.Get<PositionComp>().Value);

    for (int i = 0; i < 15; i++)
    {
        UpdateSystems(DeltaTime);
        yield return null;
    }

    Assert.True(Vector2.Distance(Player.Get<PositionComp>().Value, ...
        Enemy.Get<PositionComp>().Value) < 5);
}
```

Code 7.3: Example physics unit test

7.2 Design Choices

7.2.1 Logic in Components

As a general rule, it's best to leave out any form of logic from components. Components should only hold data, either intrinsically as a flag, or with data fields. Introducing any form of logic inside them will often limit usage or make it more complex.

That being said, we noticed some kind of logic can be introduced without limiting the usage. We have basic general logic inside HealthComponent:

```
public struct HealthComp
{
    public float Current;
    public float Max;

    public bool IsMax
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        get => Current >= Max;
    }

    public float Percent
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        get => Current / Max;
    }

    public bool IsDead
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        get => Current <= 0;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public void Reset() => Current = Max;
}
```

Code 7.4: Basic properties in health component

This is mostly to not duplicate code. And follow the same standardized logic. And it's a nice QoL feature not having to store/ref the struct when we want to find any of these three conditions.

Chapter 8

Performance

In this chapter we look at performance of ECS relative to other programming patterns, and on our game's performance in general.

8.1 Pattern Benchmarks

To test for performance differences, we have made a few small benchmarks to test the performance of ECS compared to other programming patterns. We could not use the game that we have created as a benchmark, because it would take far too much time to develop a full-scale game multiple times in different programming approaches. We will discuss performance of our main thesis game later on in this section.

There are the four small benchmarks we created:

- ECS benchmark: uses ECS technology that we have been researching in this thesis.
- EC benchmark: uses basic entity-component pattern.
- Unity EC benchmark: uses Unity GameObjects, that have a great deal of overhead.
- OOP benchmark: uses more object oriented approaches, like inheritance.

In those benchmarks we have measured how many units the systems can handle before the CPU cycle time was longer than 16.67 milliseconds (60 frames per second). There were no graphics included in the benchmarks, so GPU performance was not a part of a benchmark, as it is not as relevant to ECS. We also ignore the initialization and object creation times, so only entity iteration is being measured. Benchmarks work in a way that they add more entities to the simulation in every iteration until frame time exceeds 16.67 ms. At that point benchmark stops and prints the number of entities.

We have used 2 PC stations for our benchmarking:

- PC1: Intel Core i7-4790 CPU with stock (unmodified) clock speed 3.60 GHz, 16 GB RAM DDR3 1600 MHz, Arch Linux linux-lts 6.1.21-1.
- PC2: Intel Core i5-12600k CPU with stock (unmodified) clock speed 3.70 GHz, 32 GB RAM DDR5 4800 MHz, Windows 10 21H2.

We're only measuring iteration time and skipping in the benchmarks, for a few reasons. First and foremost, object initialization usually isn't a performance bottleneck, because of object instance management concepts: pooling, non-lazy pre-initialization, or careful memory management. On the other hand, iteration doesn't have any of these tricks. Second, object updates usually take a certain amount of time and occupy the vast majority of CPU frame time. So by optimizing objects update cycles, we're optimizing 80% of the game's frame time.

The only exception to not entirely ignoring initialization times are Unity benchmarks, we can't only measure iteration of Unity behaviors, since we don't have direct control over them. So instead we measure the initialization time, and subtract it from total CPU frame time. It's not perfect, but it's one of the better approaches when we don't have direct control over objects lifetimes and update cycles.

Our second set of benchmarks is about very simple timers, with intervals and a counter. It's to stress test how many each pattern can handle, their design, and how they work together with concurrency.

It's very rare that a game needs hundreds of thousands or millions of entities. But there are some video game genres where it's normal to have at least a few thousands entities being updated at once. In these games, it's very important that these entities have very low general overhead. Many Real Time Strategy (RTS) and general strategy games have thousands of entities updating every frame. So it's very important that the basic functionality that is updated constantly, doesn't take too much CPU time.

This lack of optimization can be even noticeable at lower entity counts. One example of this is an RTS game, StarCraft 2. With only 500-600 active fighting entities, the game starts to slow down, and lag. This is most likely to the nature of how the game was designed initially. The developers didn't forsee and design around big amounts of entities all being active at once. So now the initial design is restricting the applications performance.

8.2 Results

Here are our results. Bigger number is better. Number indicates how many entities could be made before frames per second dropped below 60. Result written in a table is an average of few benchmark runs.

	Basic Follow PC1	Basic Follow PC2
ECS	224 000	263 000
Unity EC	7 600	22 250
EC	216 000	255 000
OOP	217 000	258 000

	Timers PC1	Timers PC2
ECS	6 300 000	12 100 000
ECS 4 threads	12 700 000	35 500 000
Unity EC	25 000	116 700
EC structs	4 200 000	8 400 000
EC classes	1 600 000	2 300 000

EC Time benchmark struct version perfectly shows why sequential memory layout is so important. In fact, by default the class version had sequential memory, because we instanced new versions of classes after each other. But this is not the usual way of how objects are created, same instances of objects aren't instanced in bulk. So we fixed the benchmark by adding padding.

The timers benchmark is missing the OOP benchmark for a reason. EC and the OOP version would be essentially the same.

8.3 Performance of our game

Profiling findings were all done on PC1.

We profiled that our core systems take on average about 0.8ms CPU time. While the entire application took around 2.6ms CPU time on average. This results in stable 400 FPS.

On standalone builds, the game took around 260MiB RAM. And 93MiB of space, uncompressed, and 80MiB of space compressed.

While on the WebGL build, the game took 14.2MiB of space, uncompressed.

The biggest performance reducing factors were the visuals, more precisely one visual feature. Damage numbers that are generated with a mesh when the player deals damage to enemies. Turning this feature off, reduces the average frame time from 5ms to 0.8ms, an effective 600% increase in FPS.

The biggest performance factors for our core systems, were the systems that work every frame on enemies. Animations and visuals like: "EnemyRenderingSystem", or "ShadowRenderingSystem". That being said, both of these only took about 0.25ms average CPU time in the stress test.

Even on stress tests, where we created big amounts of enemies, one new enemy created every 10 ms, resulting in 500 enemies every 5 seconds. The player also had all upgrades in the game, to force as many calculations as possible. The game ran rather smoothly, with our core systems taking an average of 2.7ms CPU time, and 13ms average combined time. The game was at a stable 60-100 FPS.

One of the bigger problems in the stress test were physics updates, when it ran for longer periods of time. The physics engine started to take more time processing enemy collisions.

8.4 Commenting results

ECS was the fastest in our benchmarks, but by a small margin in the follow benchmark. On the other hand, difference between ECS and other patterns in timers benchmark was quite big.

In Basic Follow benchmark ECS was faster than other benchmarks, with EC being 3-3.6% slower and OOP being 1.9-3.5% slower. This is not a significant margin for ECS, although it is worth mentioning ECS was consistently best. Unity EC was much slower than other 3, but it was expected because of limitations of this method.

In Time benchmark, which tested more logic calculation, ECS was further ahead over other patterns. EC with struct benchmark was about 30% slower, and EC with classes was 75-80% slower. Unity EC again was significantly slower again, but it was expected to be so.

Chapter 9

Discussion

In this chapter we discuss how we approached each goal, whether the goals were easy or hard to achieve and what we learned. We will also discuss what we could have approached differently or what could have been done better and what would we do if we had more development time.

9.1 Learning and using ECS, and its challenges

ECS proved to be relatively easy to learn for us. Its architecture of storing data inside components, which then are contained by an entity, and then having ECS systems iterate on queried entities is quite intuitive for both of us to use. Researching, and experimenting with ECS frameworks to find the best one for our needs was definitely a good first step. As harder-to-use frameworks would probably make ECS learning time longer, and we would be left with less time for game development. During development we did not feel DefaultECS framework lacked any features important to us, and was a perfect fit.

There were however small issues with habits made with OOP development to get used to, like keywords used inside for-loop, especially those mentioned in section 6.2.1, where we would use "return" keyword instead of "continue". This would cause the Update function inside an ECS system to be exited prematurely, thus stopping the system before it could do calculations of every entity it should. In the beginning we found ourselves repeating this mistake often when we were developing new systems, but we have overcome this habit later during development.

9.2 Process of video game creation

During the game development we did not have any issues with Unity or C#.

Creation of systems inside the game was done quite quickly, in about seven weeks. This means we were finished faster than we expected, as we did not take as much time for developing the MVP.

We had developed a roguelike video game which has plenty of systems and mechanics. Careful planning was necessary to implement many of these systems. In the beginning, we have developed a Minimum Viable Product with basic functions, like controllable player which shot projectiles around, and one type of enemy to follow player and do damage in close proximity.

Later on in the development we have focused on more complex systems. To fit in some of those systems, we had to refactor the base MVP systems.

- For continuous gameplay, we have made a system which starts a wave for 30 seconds. During a wave a certain amount of enemies would spawn, with later waves having increasingly stronger enemy types. After that time, a new wave would start with new enemies spawning in.
- We have created the upgrades system which allowed the player to buy upgrades between waves, for the gold he earned for performing well during a wave.
- The player has also gotten a secondary objective in form of a getting into certain position to spawn area-of-effect damage, which would help with defeating a wave of enemies.
- We have created some modifiers which could be added to both player and enemy entities, like damage over time, slow and evasion percentage chance.
- We created two systems for visual indication of enemy health. First one changes color its color to a less saturated version as they lose hp. The other system displays a small health bar over the player and the enemies.
- A main menu was added to allow the player to switch between game options, difficulties, map size and starting player characters. From there the player could also load the last game, as we have implemented a save system which saves the game between waves.

Some features were dropped for now, but can be implemented later on, with some refactoring being necessary. An example of that was an idea of a snake enemy, which would consist of multiple Unity GameObjects, which would form a chain with head steering the rest of the body. Our systems right now are implemented in a way that one ECS entity is tied to one Unity GameObject (which is responsible for graphics and physics), and a big refactor would be necessary to allow for that type of enemy to be implemented.

9.2.1 Thoughts about testing

The unit tests we wrote helped us with catching a few fatal bugs introduced in the development time. We think the unit tests were a worthwhile investment, and we wanted to write more of them. But due to the nature of closed game engines, and not having full control over all lifetimes and update times, we weren't able to write many more game mechanics tests.

The best part of unit tests were that when we added a new feature, which interacted with previous features in some way or another. We could for the most part simply compile the new scripts and run all of the unit tests to check if anything became broken. This decreased the burden of testing on us.

Next time if we chose an open source engine, with code availability to physics and update times. We will most definitely put even more development time and care towards unit and integration tests.

9.2.2 Creating assets

No person in the team had any real arts background or experience, so when creating sprites and art for the game we have used Gimp, but more than a half of art in the game are open-licensed textures.

To create sound effects for the game, we had to learn how to mix sound. Ableton Live 11 and Audacity software was used for this task. Process of creating the sound effects consitent of finding royalty free simple sound packs and mix kits. Then multiple sounds with effects like reverb, pitch or tempo change and equalizers were merged together in trial-and-error process, until we liked the final result.

While we did most of the work and development ourselves, there are few things we really could not produce on our own. An example of it is the music soundtrack in our game, to which we have bought rights to use.

9.3 Problems during development

During development we have encountered a few bugs, smaller and bigger ones. This is completely normal in software development process. Here we will describe what issues came up and how we fixed them.

9.3.1 Wrong GameObject reference

An example of an issue we had early in development happened when in EnemySpawner, we were assigning a wrong Unity GameObject to Enemy Entities. This resulted in a situation where if a player shot an enemy, a different enemy would disappear from the game, and the one that was supposed to die and disappear would stay.

This was a simple bug that had a typo, and saved the wrong entity reference to the component.

9.3.2 Color health indicator on enemies

After adding the system which lowered the color saturation of enemies when they had low health points, we started to get enemy color mismatch when taking damage. Some enemies had wrong colors on full health, since we had multiple other systems that were also manipulating their color values, bringing its value up and down.

The solution was to save the original color, and apply it back if the entity became full health.

9.3.3 Unit tests

Testing using unit tests which used graphics and systems that don't do anything with game logic, caused unit tests to break, since the unit tests are ran in editor mode, not runtime mode. This means that no runtime engine code is being executed, making it so the unit tests fail because of null reference exceptions. The solution was to save which systems are visual only, and disable them when The unit tests start.

9.3.4 Ahead of Time compilation

One big technical problem was with Ahead of Time (AoT) compilation in Unity. We are targeting the WebGL build (so the game can be run in a browser), we cannot compile our code into Dynamic-Link Library (libraries of code and data), as browsers cannot use DLLs. Instead we need to use ILL2CPP. The problem was, that it uses AoT compilation. And we use attribute filter creation for querying entities in a system (example: Code 4.1), making it so the compiler can't AoT compile our filters, breaking the build.

The solution was to force the compiler to compile code for all the possibilities with the filters. So we made a static class with a static method that tells the compiler to generate all the AoT code for these filters.

```
internal static class BuildAOT
{
    private static void AOTCodeGeneration()
    {
         // All components registered for AOT compilation
         // This is required for iOS and WebGL builds

         AoTHelper.RegisterComponent<VelocityComp>();
         AoTHelper.RegisterComponent<PositionComp>();
           ...
     }
}

public static void RegisterComponent<T>()
{
    static bool Filter(in T _) => true;

    Filter(default);

    new EntityQueryBuilder(default, default)
        .With<T>().WithEither<T>().Or<T>().WithEither<T>().With<T>()
        .With<T>(Filter).WithEither<T>().With<T>(Filter)
        .Without<T>().WithoutEither<T>().Or<T>().WithoutEither<T>()
        .Without<T>().WhenAdded<T>().WhenAddedEither<T>().Or<T>()
        .WhenAddedEither<T>().WhenAdded<T>().WhenChanged<T>()
        .WhenChangedEither<T>().Or<T>().WhenChangedEither<T>()
        .WhenChanged<T>().WhenRemoved<T>().WhenRemovedEither<T>()
        .Or<T>().WhenRemovedEither<T>().WhenRemoved<T>();
}
```

Code 9.1: AoT compilation of ECS filters

9.3.5 WebGL saves

Because of how game platforms store their games, in a non-static internal url, the save directory for the Unity Engine is different each build. This means that we couldn't use the built-in Unity way of saving our save states. So we had to improvise and instead of letting the game engine figure out the save directory, we use a custom static one for web builds.

```
private static readonly string defaultSavePath
#if UNITY_WEBGL && !UNITY_EDITOR
     = "idbfs/" + Application.companyName + "/" + ...
        Application.productName + "/saves/";
#else
     = Application.persistentDataPath + "/saves/";
#endif
```

Code 9.2: Save directory path dependent on platform

9.4 Comparing our game to other roguelikes

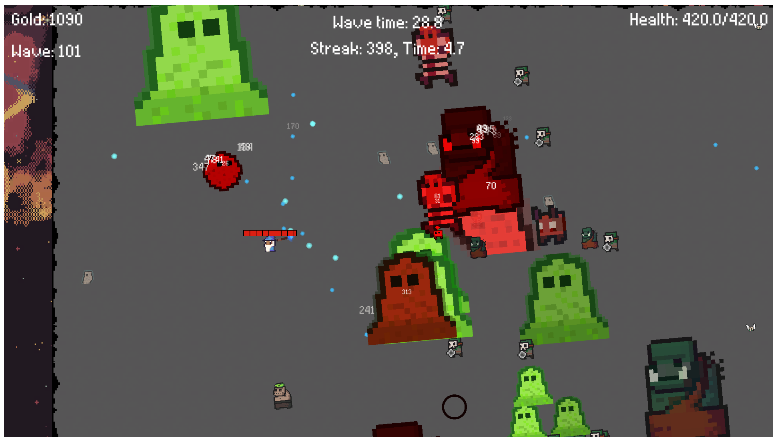


Figure 9.1: Gameplay of our game

In this section we compare our game to other roguelikes. Some aspects can be hard to compare, as opinion of a game is based on how a player feels during playing, but we can compare which games has more content like enemies or different side mechanics.

In development of our roguelike game we have put focus on replayability. This was achieved by implementing multiple upgrades, varied enemies, having an infinite gameplay loop and different difficulty modes.

- Multiple upgrades: we have 4 common 7 uncommon and 28 rare, for 39 total upgrades. Upgrades provide randomness to the game, since player gets only 3 available upgrades to choose from each round, and every run can be slightly different because of it.
- Varied enemies: a player in our game can find 20 enemy types. The enemies vary in their statistics, behavior and interactions between other enemy types.
- Infinite gameplay loop: we have made 20 hand crafted waves, but there are more generated waves afterwards which can be played for a longer time. After finished run, the player can start another one which will not be the exact same.
- Difficulty modes: we have few different modes with varied enemy numbers and other statistic changes, so that a player can make his experience easier or harder.
- Easy to play, hard to master: A player is rewarded some amount of gold he can spend on upgrades. To maximize that amount, the player has to learn to avoid being damaged and keep the kill streak up.

9.4.1 Comparison with The Binding of Isaac: Rebirth

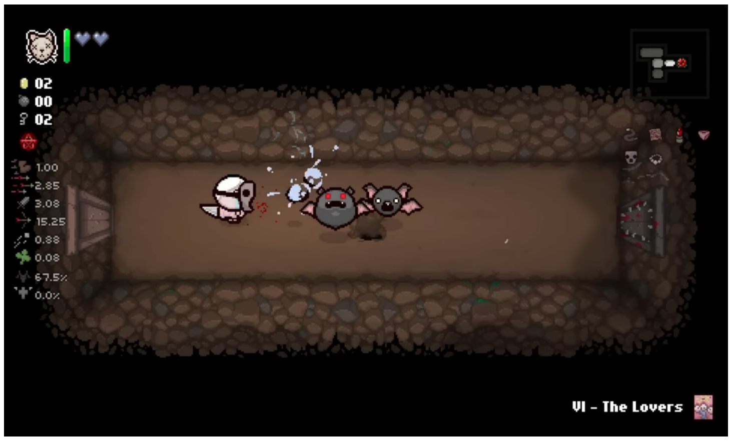


Figure 9.2: The Binding Of Isaac gameplay

The Binding of Isaac is a established roguelite video game. The basic idea of it is that a player walks between rooms in a random dungeon. Each room contains a challenge in form of enemies or environmental traps, and the player has to kill all of the enemies to proceed to the next room. A player has to beat all dungeon floors to win. The game implements random upgrades by giving items to the player, each having a unique random effect. The boss type on each floor is also randomly selected.

The first thing that gets notices when comparing our games is that The Binding of Isaac has a good looking and consistent artstyle. We could not achieve this kind of artstyle without having a dedicated artist in our team. Other things coming to mind is more varied upgrades. In Isaac most upgrades come with visual change in addition to statistical or behavioral change. Our common upgrades often come only with a statistical change like increased movement speed. Another thing is that Isaac has much more content. The game has been out since 2014 and has since got 3 major expansions. Inside the game there are 196 different enemies, about 716 items, which is much more than what our game has, but it was expected as the game exists for about 8 years. Isaac has also more side mechanics like varied shops and hidden treasures. Our game has some side goals like hitting the circle effect on an optimal timings and optimizing gold in a round, but isaac has more. Isaac is also more resource oriented, the player has to make decisions about how his resources will be used.

9.4.2 Comparison with Vampire Survivors

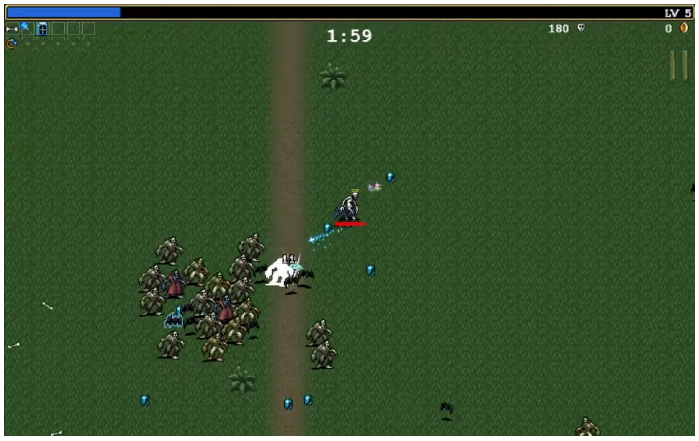


Figure 9.3: Vampire survivors gameplay

The Vampire Survivors is one of the latest hits of roguelite genre. The goal of the game is to survive on a large, open arena for 30 minutes. The player has to fight large amount of enemies, with enemies becoming harder with time. Defeated enemies drop gems, which can be later used to buy weapons which work as upgrades for the player.

Vampire Survivors has 198 enemy types and 73 weapons. There are 12 maps which serve as different difficulty modes (we have 4 difficulty modes). Inside the game there are on-level merchants, to which we do not have an equivalent mechanic in our game, as previously mentioned our game could use more side goals. Gameplay in our game is similarly placed in a large arena, but we have a different upgrade mechanics. Vampire Survivors limit the player to having 6 weapons at a time. Vampire Survivors and our game both have a system which allows better players to get more game currency. In our game it is done mostly by not being hit and making a large killstreak, in VS it's done by picking up gems from defeated enemies, which can be hard because of the other enemies around.

9.4.3 Norwegian Game Awards

During the process of creating this bachelor, we have submitted our game for participation in Norwegian Game Awards gamejam. We have not won any awards, but we are satisfied with how our game looks and plays relative to other games in that competition. Other teams in there have been bigger in size, ranging up to 30 people (including artists), and the size of our team has put some limits at what we could achieve.

9.5 Performance and benchmarking results

In this section we will discuss process and results of performance measuring and benchmarking from the chapter 8. We will look at what systems are the most performance heavy in the game, and how the programming patterns have performed in the benchmarks.

9.5.1 Game's performance

Overall, when only looking at the game's logic. The performance of the ECS systems was superb. The game can be ran smoothly on very low end hardware, when extra visual effects like damage numbers are turned off. While an average PC will have very good performance even with those extra visual features.

9.5.2 Pattern benchmarks

It is hard to compare programming patterns between full scale games, since we would have to develop a few games simultaneously.

The bencharks we have made are based on simplified scenarios, which are not completely representative on how much logic and calculations there are in full-scale games and do not necessarely show results as what real situation in a full-scale games would have shown. For example, these benchmarks do not take GPU performance into consideration, since we only evaluate ECS and other patterns and their performance on CPU, while the process of rendering an image by a graphics card is beyond the scope of this thesis. When it comes to measuring real performance of games, GPU performance is often a factor as important as CPU performance.

Overall, the benchmarks show that all three ECS, EC, and OOP patterns can be very performant, if handled correctly. But OOP is often not programmed this way in SoA.

In general structs should almost always be faster than classes, because of reference and value type semantics. In our benchmark, the difference between structs and classes is rather big, because of sequential internal memory in RAM. Because of this sequential memory, the CPU prefetches the correct next data, speeding up the iteration process. If our benchmark used more nested classes, the performance would worsen even more.

The ECS benchmarks came out as expected, being the fastest ones. Same can be said for the ECS benchmark with multithreading. Since the data can easily be split into multiple thread workers, it gave us significant performance improvements.

Unity EC benchmark has a significantly lower score than other benchmarks, but it does not mean this method has no practical uses, as it is fairly easy to learn and has many learning resources found online. It may be used in prototyping and sometimes for finished product as well, if it is not too resource demanding on a PC. This method may be the first one to be learned by aspiring game developers, who do not have any software engineering background.

9.5.3 Challenges with benchmarking and comparisons

Because we ran unit tests in a runtime/production scenario, it meant that we couldn't run hundreds of these benchmarks and find a great average. This approach has its upsides and downsides. The biggest upside being that the benchmark is as close to the real world scenario as possible. Being ran at runtime, and being measured every frame until a certain threshold is met, makes for a great real scenario. While benchmark that run in editor non-runtime mode, would be simulated, have better averages, much faster to perform, but less accurate to a real scenario.

Unity Monobehaviours

We didn't have full control over MonoBehaviour GameObject in Unity. Which means that we couldn't directly measure their update time. This introduced a problem, on how to compare the Unity benchmarks with ECS benchmarks.

The solution we came up with, was to instead to measure initialization time for the MonoBehaviour GameObjects for the Unity benchmark. And subtract that time from the total delta time. Resulting in mostly the time it took for the benchmark scripts to run. This approach isn't perfect, since other logic might take some time, but they're very minuscule, and shouldn't affect the benchmark results by a large margin.

Classes and structs

Another interesting issue that came up, was that somehow class version was as fast as the struct version in the Time.EC benchmark. Which was a very peculiar result, that at first didn't make sense to us. After a quick round of thinking, we hypothesized that the class instances are created sequentially in memory, since they're created after each other. Making the CPU hit rate as high as the struct version.

This was fixed by adding padding between the class instance creation, by creating new object instances in the loop, the objects weren't as sequentially spaced. Making the benchmark around three times slower than the struct version.

9.6 ECS relevancy

Games are a bit of a special medium when it comes to development. Game developers have tried, and still are trying to apply the standard software patterns to games.

But since the early 2010's new pattern has been developed, ECS. A lot of newer big scale games have embraced this pattern, like Frostpunk [4] and Overwatch [5].

In our benchmarking scenarios ECS proved to give better performance for up to 80%, but sometimes it was barely few percent better. It was however consistently the best performing pattern.

Learning the ECS pattern came for us without any challenges. We think that the time factor of having to learn new architecture when coming from OOP is minimal

During development we felt that we can implement new ideas quickly, which usually was done by creating new components and a system which would work on data from those components. Development time was relatively quick thanks to decoupled and flexible code, which allowed for adding new systems without refactoring other systems in most cases.

In our opinion it's also much easier to code for concurrency in ECS, while that's not really the case in other patterns/concepts. Making it the superior pattern of choice for a lot of video games. This is because as described in the section 4.1.4, the developer can select independent systems and separate them into processes. Separating logic in other programming patterns can be a much harder task.

There are situations where ECS is not an optimal pattern. Typical software does not need

ECS because it does not have systems that run every frame, same data types and similar behaviors like most video games have. In those scenarios OOP is still probably the most optimal choice.

9.7 Project Management

As noted earlier in section 6.1.1, instead of using kanban boards or scrum sprints, we opted for a more minimalistic approach. This approach worked for us, since we were a small team of two developers, so everything was handled very smoothly without any friction. If given a chance to approach another project with this type of management, we would definitely try it again, as we are happy with the results.

That being said, this approach is most likely not an adequate one if the team is larger than two people, or if the areas of development work are split. This is because both of us could have done each other's work for the most part. We easily cooperated on multiple diverse tasks. This might not be possible in more specialized development teams.

Chapter 10

Conclusion

In this chapter we summarize what we have learned and talk whether we achieved the goals we have aimed for.

10.1 Learning ECS

The first objective was aimed to learn the theory behind the inner workings of ECS and its practical use. We have achieved this by doing a thorough research on how ECS works, and research on different ECS frameworks available, which allowed us to choose the right framework for the project. In chapter 4 we explained the differences between ECS and other patterns, and went over the potential problems in this pattern, as well as potential solutions to them. In later chapters we have learned and described practical uses of ECS.

10.2 Video game

The second objective was to create a roguelike video game. We have achieved this by developing a complete roguelike game, that is playable both on a standalone build and on a web browser on a large range of devices. We are satisfied with the frame time performance of the game. During this development we have learned how to apply ECS solutions and architecture into real production problems.

10.3 Relevancy of ECS

The third objective was to decide if the ECS pattern is a relevant technology that is worth investing time in by the developer. Our research of the ECS pattern suggests that ECS can be an optimal choice in many video game development scenarios. The ECS pattern allowed us to write clean code and a easy to understand architecture of the game's systems. The performance achieved in ECS was consistently better than other patterns in the benchmarks, but sometimes by a small margin. The ECS pattern especially excels in the video game industry, but it may have a limited use outside of it.

10.4 Further development

At this point our game is a standalone roguelike experience.

We could however continue working on its performance by further optimizing critical systems, or we could introduce some new features like graphics menu where player could switch-off non gameplay-essential features like damage numbers.

We could also continue working on playtesting and tweaking numbers or creating more content in the game, like more secondary mechanics, upgrades and enemies.

Bibliography

[1] Areej. Difference between l1, l2, and l3 cache: How does cpu cache work? https://www.hardwaretimes.com/difference-between-l1-l2-and-l3-cache-how-does-cpu-cache-work/, 2021.

[2] Sheridan College. Evidence that viewers prefer higher frame rate film. https://source.sheridancollege.ca/cgi/viewcontent.cgi?article=1000&context=centres_sirt_works, 2015.

[3] Epic Games. Unreal engine 5. https://www.unrealengine.com/en-US/unreal-engine-5, 2023.

[4] Aleksander Kauch GDC. Content fueled gameplay programming in frostpunk. https://youtu.be/9r0tJCUDjtQ?t=192, 2021.

[5] Tim Ford GDC. Overwatch gameplay architecture and netcode. https://youtu.be/zrIY0eIyqmI?t=95, 2017.

[6] git scm. Git. https://git-scm.com/, 2023.

[7] Intel. Memory performance in a nutshell. https://www.intel.com/content/www/us/en/developer/articles/technical/memory-performance-in-a-nutshell.html, 2016.

[8] Paillat Laszlo. Defaultecs. https://github.com/Doraku/DefaultEcs, 2022.

[9] Paillat Laszlo. Ecs.csharp.benchmark. https://github.com/Doraku/Ecs.CSharp.Benchmark/, 2023.

[10] Arm Limited. Difference between l1, l2, and l3 cache: How does cpu cache work? https://developer.arm.com/documentation/den0013/d/Caches/Cache-performance-and-hit-rate, 2023.

[11] Sebastiano Mandalà. Svelto.ecs. https://github.com/sebas77/Svelto.ECS, 2023.

[12] MasterClass. Guide to frame rates: How frame rates affect film and video. https://www.masterclass.com/articles/how-frame-rates-affect-film-and-video, 2021.

[13] Microsoft. Readonly references. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-7.2/readonly-ref#readonly-structs, 2021.

[14] Robert Nystrom. Building games in ecs with entity relationships. https://gameprogrammingpatterns.com/component.html, 2011.

[15] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 1 edition, 2014.

[16] Igor Boyko Oleg Morozov. morpeh. https://github.com/scellecs/morpeh, 2023.

[17] Leopotam & Lukasz Pietkiewicz. Leoecslite. https://github.com/LeoECSCommunity/ecslite, 2022.

[18] Lukasz Pietkiewicz. Csharpecscomparison. https://github.com/Chillu1/CSharpECSComparison, 2022.

[19] Ben Stegner. What are roguelike and roguelite video games? *MakeUseOf*, 2021.

[20] Unity Technologies. Creating and using scripts. https://docs.unity3d.com/2020.1/Documentation/Manual/CreatingAndUsingScripts.html, 2021.

[21] Unity Technologies. Building games in ecs with entity relationships. https://ajmmertens.medium.com/building-games-in-ecs-with-entity-relationships-657275ba2c6c, 2022.

[22] Unity Technologies. Entities. https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/index.html, 2023.

[23] Unity Technologies. Physics. https://docs.unity3d.com/Manual/PhysicsSection.html, 2023.

[24] Unity Technologies. Unityecs. https://unity.com/ecs, 2023.

[25] John Terra. Entity component system: An introductory guide. https://www.simplilearn.com/entity-component-system-introductory-guide-article, 2023.

Appendix A

Source code and documentation

Source code is available at `https://github.com/Chillu1/RoguelikeEcsBachelor`.

All ECS logic is inside of "Source/Assets/Scripts/Core/Ecs" directory. The rest of the scripts, like benchmark scripts and non-ECS scripts are in the Source/Assets/Scripts/" directory.

To open the Unity project of the game:

- Install Unity Hub from `https://unity.com/download`.
- Install the Unity version 2021.3.16f1 from the Unity Hub.
- Inside the Unity Hub click "Open", and then select the "Source" folder inside repository.
- Open the project in Unity Hub.
- To start a game, select a "MainMenu" scene in /Assets/Scenes in project tab (bottom left corner) in Unity Editor. Then click the start symbol above the scene window.

The game was build in Unity 2021.3.16f1, and should be build on that version as well.

Appendix B

Game

Online builds of the game are available at: `https://chillu.itch.io/pixelwizardarena`, and `https://chillu1.github.io/RoguelikeEcsWeb/`.

Local builds of the game are in the "Builds" directory, split by platform.

To run the standalone builds, simply run the windows, or linux executable.

To run the WebGL build locally, a simple http server is needed. With commands like: "http-server -p 8000" or "python -m http.server"