BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

RPG GAME IN UNITY WITH PROCEDURAL ELEMENTS

RPG HRA V UNITY S PROCEDURÁLNÍMI PRVKY

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **SAMUEL LÍŠKA**
AUTOR PRÁCE

SUPERVISOR **Ing. TOMÁŠ MILET, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Líška Samuel**
Program: Informační technologie
Název: **RPG hra v Unity s procedurálními prvky**
RPG Game in Unity with Procedural Elements
Kategorie: Počítačová grafika

Zadání:

1. Nastudujte herní engine Unity, procedurální generování a tvorbu her.
2. Navrhněte hru a herní mechaniky s procedurálními prvky.
3. Implementujte navrženou hru v Unity.
4. Uživatelsky hru otestujte a zhodnoťte výsledky.
5. Vytvořte demonstrační video.

Literatura:

- Gregory, Jason. *Game engine architecture*. crc Press, 2018. ISBN 1351974289, 9781351974288
- Bishop, Lars, et al. "Designing a PC game engine." IEEE Computer Graphics and Applications 18.1 (1998): 46-53.
- Adams, Ernest, and Joris Dormans. Game mechanics: advanced game design. New Riders, 2012. ISBN 0321820274, 9780321820273

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2 a kostra aplikace.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Milet Tomáš, Ing., Ph.D.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 1. listopadu 2021

Abstract

The main objective of this thesis is to create 2D top-down RPG game with a focus on procedural generation in Unity. This thesis contains a summary of information about videogames, procedural content generation, game engines, and Unity itself. This thesis also contains solution design and implementation of the game. Perlin noise and its processing into the biome with the usage of Whittaker diagram has been used. Multiple systems to enhance gameplay are described as well. Lastly, this thesis contains testing and evaluation with a small survey.

Abstrakt

Hlavnou úlohou tejto bakalárskej práce je vytvoriť 2D RPG s procedurálnymi prvkami, ktorá má pohľad z vtáčej perspektívy. Hra je implementovaná v hernom engine Unity. Obsah tejto práce je zložený z teoretických informácií o videohrách, procedurálnom generovaní obsahu a informáciach o herných enginoch a Unity. Práca taktiež obsahuje návrh riešenia a implemetačnú časť hry. Na procedurálne generovanie sveta bol použitý Perlinov šum a jeho následné spracovanie pomocou Whittakerovho diagramu. Práca popisuje viacero systémov ktorých ulohou je vylepšiť pôžitok zo samotnej hry. Na záver práca obsahuje testovanie a zhodnotenie pomocou krátkeho dotazníka.

Keywords

game development, computer game, procedural generation, RPG elements, Unity, Game, C#, 2D, navmesh, AI, Perlin Noise

Kľúčové slová

herný vývoj, počítačová hra, procedurálne generovanie, RPG prvky, Unity, Hra, C#, 2D, navmesh, umelá inteligencia, Perlinov šum

Reference

LÍŠKA, Samuel. *RPG Game in Unity with Procedural Elements.* Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Milet, Ph.D.

RPG Game in Unity with Procedural Elements

Declaration

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

.....
Samuel Líška
May 9, 2022

Acknowledgements

My deepest gratitude belongs to the thesis supervisor Mr. Tomáš Milet. The completion of this thesis would not have been possible without his guidance, suggestions and great consultations.

Contents

Chapter 1

Introduction

Motivation

The games industry has never been larger and it constantly grows. Since computers started to be more accessible to the public, games started to show their potential. Computer games do not represent only mindless punching buttons to achieve the highest score anymore. The technology they are built on has come much farther and has transformed into beautiful works of art with breathtaking stories and audio-visual aspects. Games no longer require huge development teams to deliver. In recent years, small teams with only several people, are able to create amazing products. This development is known as indie development. Indie game development is at its peak and multiple game titles have proven that in recent years.

Goal

The goal of this thesis is to create a 2D RPG game. RPG stands for a role-playing game in which participants adopt the roles of imaginary characters. The game consists of two different parts. The first is procedural generation, and the second is the RPG side of the game. Both of these are completely different challenges and extend games‘ potential in a different ways.

Creating large worlds, filled with plenty of enjoyable content is often a big challenge for the game development industry. No doubt the best results are done by crafting everything player encounters by the hand. However, this approach requires enormous time and usually a great number of people to work on them. The result, however, will always be some kind of finite challenge for the player, that will result in repetition. Game with this type of world must focus on other aspects, such as a rich story or mind-blowing gameplay. Another way to tackle this problem is to let the computer create content for players. This method is referred to as procedural content generation, and it is the fundamental concept of this thesis.

Another aspect implemented in the game, are simple RPG elements. These elements might be implemented in various ways. From creating player‘s own look of character, skill trees that allows the player to determine his path, to experience a system that influences the player‘s capabilities. But the main idea stays the same. The player is able to interfere with the world and character, in which he wanders in.

For creating games, game engines have been developed to make their development easier. There are several game engines such as Unreal Engine, CryEngine, Godot, Unity engine, and many more. Unity is the perfect engine for indie developers, it is very popular and has an enormous community base, and that is the choice for this thesis. A more detailed description of the game is given in the overview chapter 2.

Chapter 2

Overview

This chapter roughly describes the game itself. To outline what the actual game consists of, some of the most popular games and their mechanics that inspired this thesis are described here. Some concepts and game mechanics are inspired by the games mentioned below, while others expand these ideas and might not be that obvious. Also, multiple genres, and their features and possible integration are mentioned here.

The game is an uncompromising RPG game full of action, combat, and challenge. With the map being procedurally generated, each playthrough slightly differs in terms of map design. The goal of the game is to find and complete four missing keys, hidden within crystals located throughout the map. This is known as **The Main Quest**. The player can encounter multiple enemies, each being slightly different in terms of speed, attacks, and overall strength. Game rules and crucial information about the gameplay are slightly presented in the game presentation 2.2 chapter and described more in detail in the solution design chapter 3.3.6.

Figure 2.1: Player wandering it the Forest biome. Healthbar in the bottom left corner indicates low health.

2.1 Games with similar concepts

All of the games described below shaped how the final product was intended to look. Several games influenced the idea, however, their influence was minor compared to these titles.

2.1.1 Diablo

Diablo is an *action role-playing*[1] video game developed by Blizzard Entertainment in 1997. The player moves and interacts with the environment mostly by the usage of a mouse. Character's capabilities, such as casting a spell are performed on a keyboard. Player can acquire items, gain experience to learn new spells, defeat enemies and interact with *NPCs* [2].

Dungeons are **procedurally generated** with specific themes for each level. What that means is, for instance, the catacombs have long closed corridors, whereas caves are not so linear. As the player enters a dungeon, a random number of quests are assigned to him. Quests are optional but usually offer significant rewards. As the player progresses, his character is becoming more and more powerful, and he is revealing more of the story. Diablo has also introduced a character classes system, that hugely affects the gameplay experience. Each class has its advantages and limitations, such as class unique items, different preferable attributes, and so on. The classes are The Warrior, The Rogue, and the Sorcerer. Last but not least *multiplayer*[3] is available for up to four players, where they can be either aggressive towards themselves or play co-operatively.

Figure 2.2: Player explores the catacombs. Level up button indicates the character has attribute points available to distribute.

2.1.2 Don't Starve

Gamespots' review [7], defines Don't Starve as an action-adventure game with a randomly generated open world and elements of survival[4] and roguelike[5] gameplay, where combat is handled by pointing and clicking with the mouse. Other activities such as usage of the toolbar are controlled by a keyboard. The goal is to survive as long as possible. While Wilson is the main protagonist of Don't Starve, he has no special abilities. Besides Wilson, there are several playable characters, that offer different limitations and benefits that change gameplay drastically.

The important part of this game is **procedurally generated world**. Want a new map? At any time there is a possibility to generate a new unique and challenging world.

[1] sub-genre of video games that combines elements of action game and role-playing genre
[2] NPC - non-player character
[3] more than one person can play in the same game environment at the same time
[4] sub-genre of action video games, usually set in hostile, intense, open-world environments
[5] sub-genre of role-playing video games characterized by a dungeon crawl through procedurally generated levels

World in Don't Starve consists of **biomes**. Biomes usually differ by the type of terrain they are composed of. Each type of biome usually contains the same type of resources and creatures, that may or may not be specific to that biome. Each world generation has a different configuration that generates different content. The complete process of world generation is quite sophisticated and is publicly available to read here: [5].

Figure 2.3: Wilson using an item next to *beefallo* creatures that are located specifically in *Savannah* biome. In the bottom is players inventory followed by equipped items, and on the left side is the game tab. Upper right corner shows day-night cycle and players survival stats such as hunger, health and sanity.

2.2 Game presentation

After the introduction of games with similar concepts, this section describes the product of this thesis. This segment acts as a showcase of the game.

The game is a role-playing genre with procedural elements. It uses a top-down perspective, which refers to a camera angle that shows the player and the area around him from above. It is sometimes referred to as birds-eye view, god view, or overhead view. A top-down perspective is commonly associated with 2D role-playing games. The player is placed in the procedurally generated world. Primary objective is to finish *the Main Quest* described below.

The Main Quest

The primary and single quest in the game is called **the Main Quest** and reads as follows: *Acquire four missing keys to finish the game. The world consists of four biomes, each differs by terrain type and creatures that occupy them. Four missing keys and four types of biomes are not a coincidence. Each key can be found exclusively in one biome, but there might be multiple biomes of the same type across the generated world, without sharing any border at all. The key itself is located in a crystal, generated somewhere in a specific biome.*

World

As mentioned before, the world is procedurally generated which means every time a new world is created, it offers a new gameplay experience. However, some rules and aspects of the world remain the same. Those are:

- **Island**: Generated world is an island. The island has a square like structure and is surrounded by the ocean that acts as a natural barrier.
- **Climate**: Heat is unevenly distributed across the island. The highest temperatures are near the equator (center of the island). As latitude increases, temperature drops. The outcome of this behavior is, that hotter biomes are distributed near the equator while colder are more towards the poles.
- **Biomes** there are four main biomes. Hot biomes are *the Jungle* and *the Desert*, cold is *the Ashland* and in between is *the Forest*. There is also *Beach* that connects the sea, coast, and inland.

Figure 2.4: Character moving across *the Ashland*, while being healed by the *heal spell*.

Figures 2.4 and 2.5 specifies two out of four accessible biomes in the game. The Ashland is usually starting biome and is located either on the north or the south of the island. Biomes and their enemies are closely outlined in its subsection in solution design here 4.4.1.

1

Figure 2.5: Border between three biomes, *the forest*, *the Desert* and *the Rainforest*. Player is fighting Skeleton entity unique for *the forest*. Red arrow above one of lock icons indicates direction towards missing key.

1

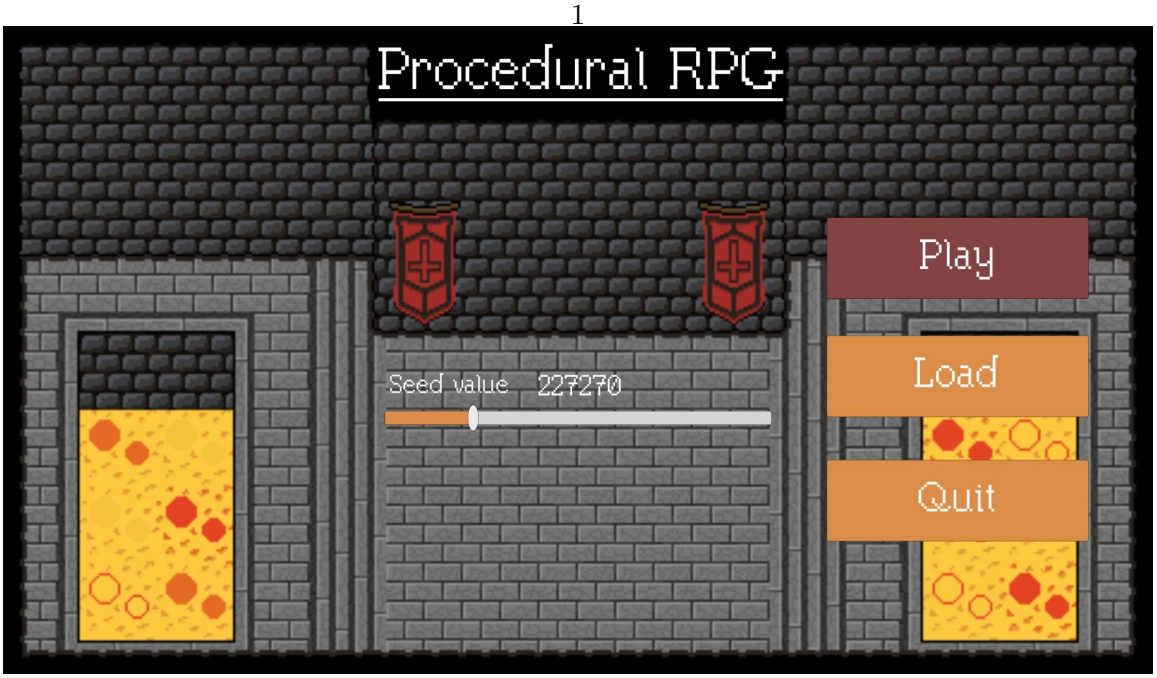


Figure 2.6: This figure shows main menu after launching the game. Menu consist of *seed* slider on the left, determining seed number to be used to generate the world, and *Play*, *Load* and *Quit* buttons respectively.

Chapter 3

Theory

This chapter consists of the description of all concepts, related to the thesis. Although video game genres were once fairly clear-cut, it is simply not the case these days anymore. There is a growing variety of genres and sub-genres to understand, especially as game developers mix and blend different types of games in new and unexpected ways. Various mechanics and concepts are described in this chapter. Mechanics such as, how randomness is implemented in the games, what is a game engine and what is procedurally generated content.

3.1 Video games

The definition of a video game can get stuck in debate as to what exactly *games* are. It does not apply just to video games but games in general. There are multiple definitions of games but let us stay with the Oxford English Dictionary definition: *Game played by electronically manipulating images produced by a computer program on a monitor or other display* [8].

A video game is a type of game that is played on an electronic device and involves user interaction with input devices, such as a joystick, mouse, or keyboard. User interaction usually creates a visual response to the player. Key components of video games are *rules*, *challenges* and *interaction* with the player. Most video games tend to feature some type of victory or winning conditions, such as a scoring mechanism, final quest, or a final boss fight.

3.1.1 Genres

Video game genres can refer to individual styles of gameplay, with a focus on different mechanics of the game. As mentioned in Video Games: An Introduction to the Industry [4]: *Most games are not pure breeds of a specific genre, but rather hybrids with many overlapping traits.* This is commonly known as sub-genre[1]. Video games and their division into genres are actually rather inconsistent. Each source of information is slightly different. This section describes some of the best-known and most common video game genres. Main source of information was book *Video Games: An Introduction to the Industry* [4] and an article *The Many Different Types of Video Games & Their Subgenres* [10] from iDTech's blog.

[1] a subdivision of a genre of literature, music, film

Action

Action games are characterized by their action in real-time, focusing on hand-eye coordination and reaction time. Genres name came directly from the fact, that player is thrown in the middle of the action. Because action games are commonly the easiest to start right away, they are among the most popular games. The genre includes a big variety of sub-genres described below.

- **Action-adventure**: Most frequently incorporate two key mechanics from two genres. Combat from action games, and multiple story-related features such as quests, items, and similar concepts from an adventure. - *God of War, The Last of Us, Tomb Raider, etc.*
- **Platformer**: Platformers get their name from the fact that character interacts with platforms, Running through obstacles, falling is actual gameplay. - *Super Mario, Donkey Kong, etc.*
- **Shooter**: Shooters let players use weapons to engage in the action, with the goal usually being to take out enemies or opposing players. Mostly categorized by player perspective, FPS, TPS[2] and top-down shooters. - *Half-life, Halo, Call of Duty, etc.*
- **Fighter**: Focused mostly on hand-to-hand combat. Most fighters have a feature of a stable variety of characters with different specializations. - *Mortal Kombat, Street Fighter, etc.*
- **Beat-em up**: Also focused primarily on combat, but instead of facing a single opponent, players face multiple waves of enemies. - *Disney's Hercules, Double Dragon, etc.*

Role-playing game (RPG)

Probably the second most popular genre predates much of what we know of modern computer games, as role-playing in a fictional world has existed for some time in tabletop forms such as *Dungeons & Dragons*. RPG games allow a player to define or alter the course of the game and the character's development, meaning RPGs often do not follow a linear story.

- **Action RPG**: Sub-genre that merges action and RPG elements from both genres. Combat takes place in real-time and is dependent on the player's speed and accuracy, while also depending on the character's attributes like strength and agility - *Witcher 3: Wild Hunt, Diablo, Mass Effect, etc.*
- **MMORPG** : Combining MMO[3] with an RPG. This genre evolved as graphical variations of MUDs[4] which were developed in 1970s. MMORPGs involve hundreds of players actively interacting with each other in the same world. Players control a character that can be improved over time as they complete missions and quests. Worlds levels and game data are persistent and are hosted on remote servers. - *World of Warcraft, Guild Wars, etc.*

[2] third person shooter
[3] massively multiplayer online
[4] multi user dungeon (usually text-based)

- **Roguelike** : Genre name based on the name of the game that inspired it. *Rogue* was a 2D dungeon crawler from 1980. The game featured a text interface and random level generation. - *Hades, The Binding of Isaac, etc.*
- **Tactical RPG**: Player more like traditional board games, where the action is turn-based rather than real-time. Tactical aspects of the game require players to use almost chess-like strategy to defeat their foes. - *Divinity Series, XCOM 2, etc.*

(a) Action role-playing game with a third-person perspective *Witcher 3: Wild Hunt* (2015)

(b) Classic FPS game with a linear story set in second world war. *Call of duty* (2003)

(c) One of the first ever 3D fps games *Wolfenstein 3D* (1992)

Figure 3.1: Figures above shows examples of very popular games in action and RPG genres respectively.

Strategy

Defined by the point of view from above, these games require players to manage the creation, collection, and allocation of resources and tactics to defeat their enemies. Gameplay is based on traditional strategy board games, and tends to give players godlike access to resources and a „god view“.

- **4X**: name is derivated from four primary goals of these games: e**x**plore, e**x**pand, e**x**ploit and e**x**terminate. Most of these have historical settings and span eons[5] of a civilization's history. - *Sid Meier's Civilization, Stellaris, etc.*
- **Real-time strategy (RTS)**: Require players to collect and maintain resources to expand bases, build armies and command units in real time. Fast tactical thinking and strategy is required to overcome enemies. - *Starcraft, Command and Conquer, etc.*
- **Real-time tactics (RTT)**: Primarily focused on the combat side of strategy, typically does not feature resource-gathering base building or economic management. - *Total War: Rome II, Desperados, etc.*

Others

There are many more video game genres in existence that are worthy of mention. Several genres that feature interesting concepts or gained huge popularity are described below. Adventure and Sandbox are the most common genres to merge with others mentioned above to create sub-genre.

- **MOBA**: Multiplayer Online Battle Arena attracts massive crowds to live events and usually has a very popular competitive scene. MOBAs might be also considered a strategy sub-genre. Players control a single character in one of two teams. Both teams try to eliminate each other to destroy the other team's base. *Dota 2, League of Legends, etc.*
- **Simulations**: All simulations are designed to emulate real or fictional reality, to simulate a real situation or event. Exists largely within the PC platform. Simulations can focus on vehicle simulation, management, construction, and even live cycles and evolution. - *Microsoft Flight Simulator, Farming Simulator, etc.*
- **Sports Games**: Extremely popular genre to simulate sports, that might be more focused on reality(simulation), of playable aspect(arcade). Can be divided into multiple groups such as racing, team sports, or sports-based fighters. - *FIFA, F1, etc.*
- **Adventure**: Genre defined by the style of gameplay, not the story or content. Players usually interact with the environment and other characters to solve puzzles and progress in the story. Genre alone is not very popular, but merging it with genres mentioned above is a very popular choice recently. - *Adventure (Atari 2600), Zork, etc.*

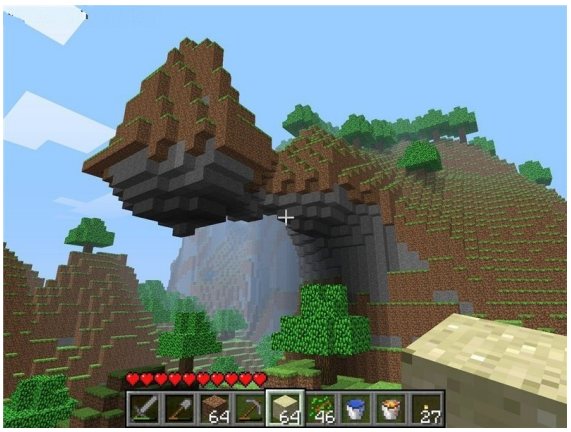
[5] very long period of time

- **Sandbox**: Sandbox or open-world, is a non-linear, free-roaming gameplay environment where the player can choose to discover and interact with any part of the gaming world. Extremely popular in combination with any of the above-mentioned genres. - *Minecraft, Terraria, etc.*

(a) Very popular MOBA game developed and published by Valve *Dota 2* (2013)

(b) Historical RTS developed by Firefly Studios *Stronghold: Crusader* (2002)

(c) 17th NHL series sports video game by EA Sports. *NHL* (2008)



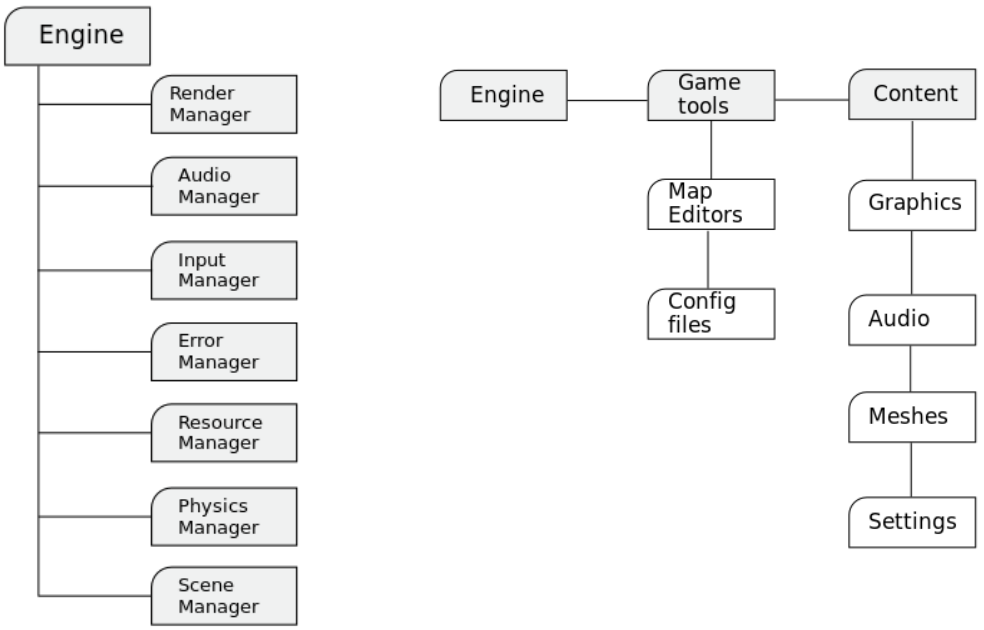
(d) Sandbox and the best selling game of all time *Minecraft* (2011)

Figure 3.2: Shows examples of video game genres mentioned above.

3.2 Game engines

According to *Oxford English Dictionary*, **Engine** is a software system, not a complete program, responsible for a technical task.

As Alan Thorn mentioned in his book, *the game engine represents everything that is abstract and applicable to all or most games* [12]. A game engine is a software designed for the development of video games. The game engine generally includes multiple libraries and support programs. The idea is that the game engine is the heart or core containing almost all the generalizable components that can be found in a game. The game engine consists of multiple essential manager components as shown in Figure 3.3



(a) Common set of managers for a game engine

(b) Game engine and its relationship to other parts of game development

Figure 3.3: Game engine managers, and relationship between game engine and other parts of game development according to *Game Engine Design and Implementation* [12]

Engines design followed by the RAMS [6] principles often benefits the game developer in many ways. Some engines are free such as CryEngine, Unity Engine, or Unreal engine whereas others might be licensed and thus unable to freely use, for example, Frostbite Engine(*EA*) or Red Engine(*CD Projekt Red*).

3.2.1 Unity Engine

Unity has been around since 2005. *It was launched in June and aimed to democratize game development by making it accessible to more developers* according to Samuel Axon [1]. Unity has been a popular choice for indie developers since then. It supports not only 2D and 3D game development but can also be applied to virtual reality. The engine is constantly maintained and updated. Unity is free software so everyone can start using it.

Unity is a cross-platform engine, meaning its products can work across multiple platforms. Unity editor is supported on Windows, macOS, and Linux. The engine itself can turn on 19 different platforms. Officially supported platforms as of *Unity 2020 LTS* according to Unity's manual page [11] are :

- Desktop platforms Windows, Mac, Linux
- Mobile platforms Android, iOS, tvOS
- Web platforms WebGL
- Console platforms Playstation, Xbox, Nintendo Switch, Stadia

[6] RAMS - essential principles of game engine design (recyclability, abstractness, modularity, and simplicity)

- Virtual/Extended reality Oculus, Playstation VR, Steam VR, Google Cardboard, Windows Mixed Reality, Holo Lens, Magic Leap

The thesis is developed in Unity Engine mainly because of its significant community, open-source, and support for indie development.

GameObject

According to Unity's manual in the GameObject section: *„GameObjects are the fundamental objects in Unity that represent characters, props, and scenery. They do not accomplish much in themselves but they act as containers for Components, which implement the real functionality.“ [11]*

GameObject has always attached *transform* component(or *rect transform* for UI), where position of the GameObject could be altered. Position could be relative to its parent, or if GameObject is parentless directly to the world space.

Components

Components define the behavior of the GameObject they are attached to. Most of the component's properties can be adjusted in the Unity Inspector itself, without interfering with the script. Although Unity's inspector is a powerful tool, with deeper adjustments scripting is a critical part of the work. Script is also considered as a component. More about components can be found in Unity's manual in section *Components* [11].

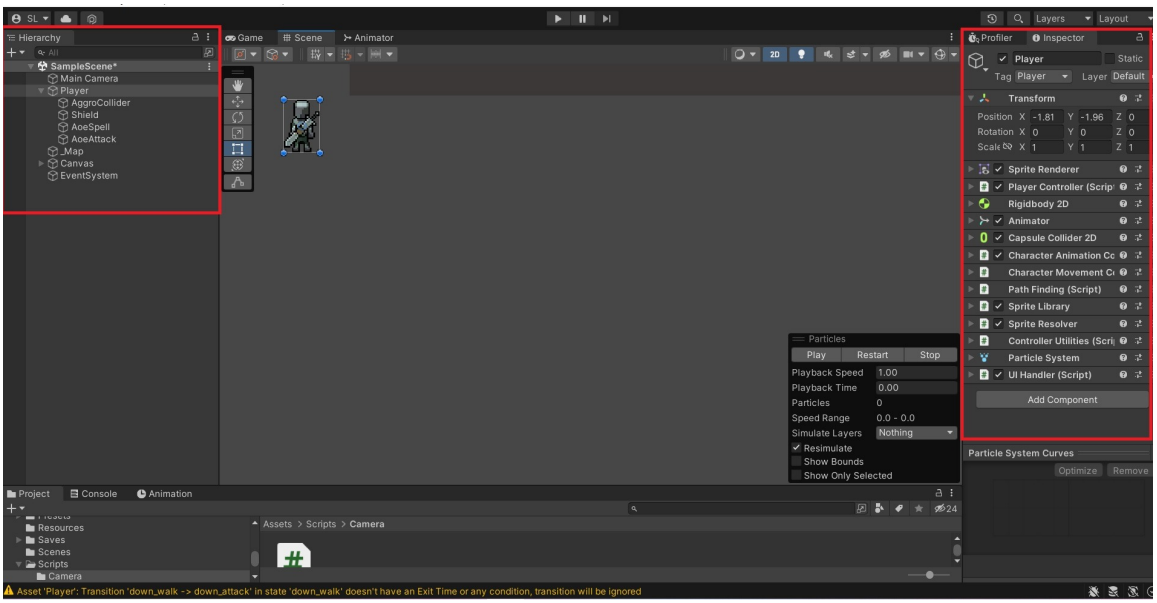


Figure 3.4: Marked on the left side are *GameObjects* in the shown scene, whereas marked on the opposite side are *components* attached to the Player *GameObject*

3.3 Procedural Content Generation

In the book *Procedural Content Generation in Games* [9], Procedural Content Generation (PCG) is defined as a **algorithmic creation of game content with limited or indirect**

user input. In other words, PCG is software that can create game content on its own, or with human interaction.

3.3.1 What is content?

Content is most of what is contained in the game, starting with levels, maps, items, music, etc. Game Engine or non-player characters (NPCs) are not considered to be content. Terms procedural and generation refer that we are working with procedures or algorithms to create something.

The most obvious reason to procedurally generate is that it removes the human element when creating content. Humans are much slower and much more expensive. PCG method makes it possible for small teams or individuals without great resources to develop great games. But PCG can open doors for completely new types of games. We have software that can generate game content at a very high speed, that is about to be consumed by the players. This content is different every time. With a proper procedural content generator, the game will be with every generation slightly different. For everyone who has been disappointed by not having more levels, and more areas to explore, this is a solution.

Newly generated content can be also tailored to each player‘s needs. Combining PCG with human interaction we can create player adaptive games.

3.3.2 Deterministic or stochastic approach

Deterministic PCG is able to regenerate the same content given the same starting point and parameters. On the other hand using a stochastic approach, recreating the same content is not possible. The generation in Minecraft is a perfect example of deterministic PCG. The world can be regenerated multiple times with a specific seed.

3.3.3 Requirements for PCG solution

The desirable or required results differ for each application. The usual tradeoffs involved are speed and quality of generated content. There are several aspects when dealing with PCG which must be taken into account [9]:

- **Speed**: Can vary quite a lot. Some content might be generated in a moment, while others might take several hours or even more. This is a huge factor when doing PCG in games. When dealing with endless worlds, content is being generated during gameplay and may outcome in stuttering or huge fps drops.
- **Reliability:** Some PCGs are capable of creating content satisfying given criteria regularly, while others might extend to the edges resulting in unsolvable, or broken content. From creating a hill that looks odd to generating a key element of the game within an area that cannot be reached, are two completely different stories. Where one is just aesthetic fault, the other one is completely breaking the game.
- **Controllability:** Generators often requires to be controllable to suit specific aspects of the content. Smoothing various aspects of content, or the opposite making them sharper
- **Expressivity and diversity:** If a generator is creating minimum differences such as changing minor aspects, the content will become tedious. Creating heavily random

content without any rules that players can learn from will result in the opposite outcome. The key is for generated content to have its *signature*. When generating a map, a player should recognize specific expressivity of content to learn from.

- **Creativity and believability:** There are multiple ways to generate content that does not look like it is procedurally generated. This is what we are trying to achieve in most cases.

3.3.4 Random number generators

Pseudorandom numbers have been used in the game industry for a long time. *From rolling dice to card games, to any other random element has been done by random number generators* (Ryan Watkins) [13]. Random element adds unpredictability to our games, and that is what makes them exciting in most cases. **PRN**s [7] and its generators are deterministic algorithms, generating seemingly random numbers. However, after some time sequence for given inputs starts to repeat. This problem is solved by altering the inputs of the generator. Input data for generators are called **seeds**, and for multiple initializations with the same seeds, output data will always be identical. This behavior may or may not be desired.

3.3.5 Noises

Noise is a series of random numbers, typically arranged in a line as a matrix. Noise functions are commonly used in computer graphics because they add random elements. In signal, processing noise is typically not desired aspect, and we want to avoid it. On the other hand, with intent to generate something that looks natural, noise is typically what we want. **Coherent noise** is a type of smooth pseudorandom noise that is generated by a coherent-noise function which has three important properties.

1. Passing in the same input value will always return the same output value.
2. A small change in the input value will produce a small change in the output value.
3. A large change in the input value will produce a random change in the output value.

Perlin Noise

Perlin noise is a function for generating coherent noise over a space. While normal RNGs [8] produce outputs that are completely independent from each other, Perlin noise generates random numbers that follow a smooth gradient. This means that for two nearby points, two similar results will be returned. Perlin noise is based on noise functions, and is created by the sum of the same function but with different *amplitude* and *persistence*. A number of noise functions used are called octaves. Perlin noise is very popular and widely used in game development. Libnoise's documentation says that key attributes for coherent noises are [3]:

- **Amplitude** - the frequency of each successive octave is equal to the product of the previous octave's frequency and the lacunarity value.

[7] Pseudo random number
[8] Random number generator

- **Frequency** - the number of cycles per unit length that a specific coherent-noise function outputs.
- **Lacunarity** - a multiplier that determines how quickly the frequency increases for each *successive octave*[9] in a Perlin-noise function.
- **Octave** - one of the coherent-noise functions in a series of coherent-noise functions that are added together to form Perlin noise. The number of octaves controls the amount of detail of Perlin noise. Adding more octaves increases the detail of Perlin noise, with the added drawback of increasing the calculation time.
- **Persistence** - a multiplier that determines how quickly the amplitudes diminish for each successive octave in a Perlin-noise function.

3.3.6 Procedural world generation

With knowledge acquired from section 3.3, it would be appropriate to describe a specific part of PCG, and that is procedural world generation. To represent the terrainheight map described below is required. To generate the height map noise functions such as Perlin Noise are used. This process basically takes coordinates and returns seemingly random values for each given coordinate.

Plain Perlin noise itself, generates rather uninteresting results. The solution is, however, very simple. Simple multiplication and addition are the keys to distinguishing and alter noise results. These are *persistence* and *lacunarity* attributes. *Persistance* is affecting length of frequency of the function, while *lacunarity* is affecting volume of function. Global attributes that affect every generator are known as **seeds**. Seeds, determines the structure of the terrain. Generated values could be altered by noise octaves, which is basically overlapping multiple values of the noise, smoothing and strengthening the result.

Heightmap

A heightmap is simply a 2D array. The advantage of a heightmap is that it is easy to implement and fast to generate. However, there are a few disadvantages, such as when there is a large amplitude, the terrain could look extremely spiky. This could, in some cases, tear down the feeling of smooth randomness. Some terrain generators use a voxel-based engine, which instead of using a 2D array, uses a 3D array. This allows for more complex structures like caves.

To popoulate the heightmap, a simple call of noise function to every single element of the array is a reasonable approach. Noise functions such as Perlin Noise or Simplex usually return float values between 0.0 and 1.0. In the heightmap higher values means bigger height.

The same approach could be used in maps representing different attributes of the terrain since noise functions offer huge variability in altering the results. Such as average temperature called heat map or average rain fall called precipitation map. Combining multiple maps could be further used to form **biomes**. More detailed information about heightmaps and their usage in the terrain generation can be found in the *Study of procedural terrain generation in plain and spherical surfaces* [2].

[9] The frequency of each successive octave is equal to the product of the previous octave's frequency and the lacunarity value.

Biomes

A biome is a region in a world with distinct geographical features, such as vegetation, temperature humidity, etc. Biomes separate every generated world into different environments, such as forests, jungles, and deserts.

Multiple aspects can have an impact on selecting the proper biome, and they vary by level of difficulty to implement. From easiest to hardest. Well, known aspects are [6]:

- **altitude:** higher altitude means lower temperatures and lower humidity. This will result in more snow-capped mountains.
- **ocean proximity:** being closer to the sea brings the temperature closer to the average temperature of the planet, and increases the humidity.
- **atmospheric circulation cells:** modeling the global winds, regional movements of the air around areas of high and low pressure.
- **negative rain shadows:** like the previous approach but with the integration of the altitude windward. This produces deserts like the Chilean desert.
- **positive rain shadows:** Take the directional derivative of the altitude and multiply it with the convolution map from the atmospheric circulation cells step. This produces forests like the Pacific Northwest.
- **ocean currents:** Fluid simulation of the ocean portions of the planet to determine the temperature of the water, sending heat towards polar regions and helping tropical areas cool off.

Whittaker diagram

Whittaker diagram requires two known aspects to determine the biome. The first is temperature and the second is humidity. As shown in Figure 3.3.6 these two aspects determine the type of a biome. To generate proper biomes with the Whittaker diagram, the key is to distribute heat and humidity values correctly. Both can be influenced by various aspects mentioned here 3.3.6.Robert Whittaker matched vegetation type to regional climate to create a triangular figure within which all biomes fall. This topic is closely described in the book by Whittaker himself called *Communities and ecosystems* [14]. Whittaker terms are:

- **physiognomy:** characteristics or appearance of ecological communities or species
- **biome:** grouping of terrestrial ecosystems on a given continent that is similar in vegetation structure, physiognomy, environmental features, and animal community characteristics.
- **formation:** a major type of plant community on a given continent
- **biome type:** grouping of convergent biomes or formations from different continents, defined by physiognomy
- **formation type:** a grouping of convergent formations

Tropical climate zones have mean annual temperatures of 20°C - 30°C and mean annual precipitation of 0 - 400+ cm. **Temperate** climate zones have mean annual temperatures of 5°C - 20°C mean annual precipitation of 0 - 300+ cm. **Boreal and Polar** climate zones have mean annual temperatures of < 5°C mean annual precipitation < 200 cm.

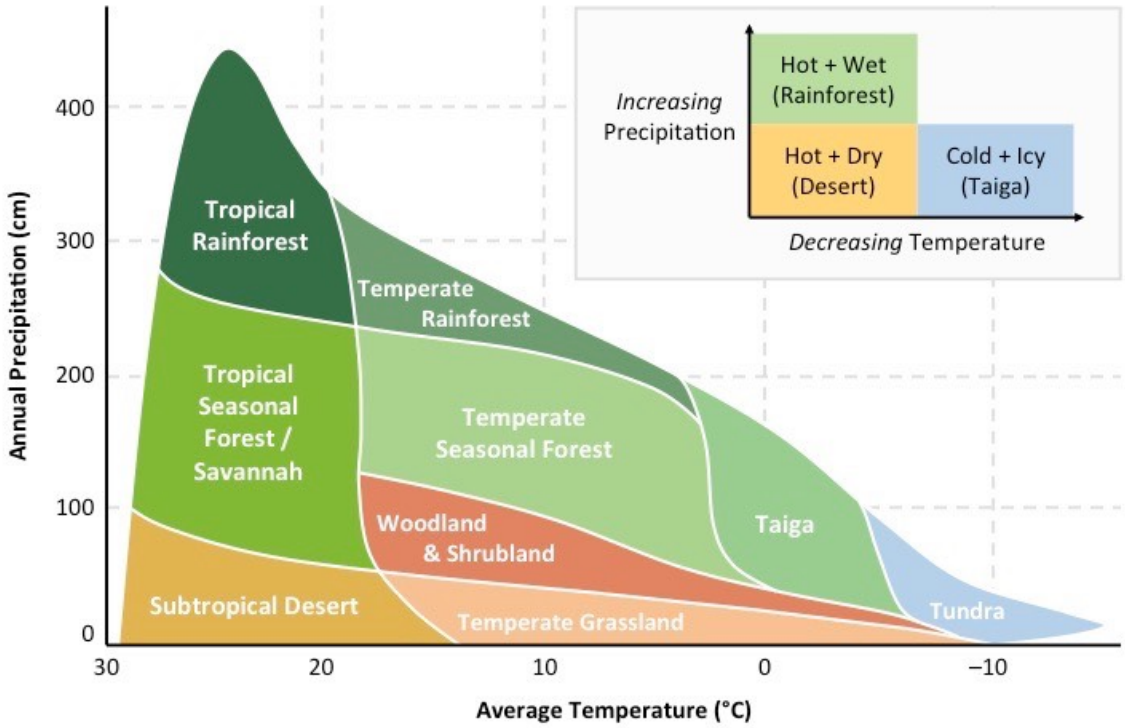


Figure 3.5: Whittaker diagram is used to determine correct biome. Clearly shows dependece between temperature and precipitation and its relation to biomes.

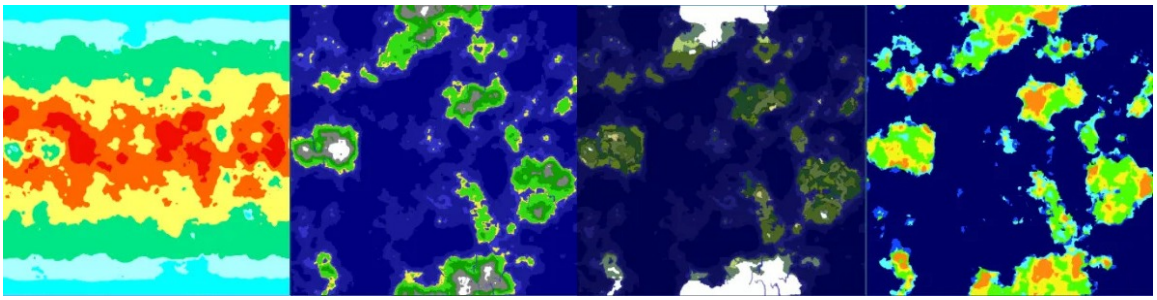


Figure 3.6: Shows heat map, height map, moisture map and biome map respectively. Resulting biome map was constructed by merging three previous maps into one with usage of Whittaker diagram 3.5

.

Chapter 4

Solution Design

This chapter is trying to elucidate to the reader the main concepts of the game. One of the most important steps is scope definition. After general clarification of the game, game mechanics are closely described. Beginning with procedural map generation, followed by character specification, enemy design winding-up with a global user interface.

4.1 Concept

In this game, the player is able to venture through the lone island deep in the ocean. The player can move and quickly discovers, that has abilities to use to defeat foes. The island has a variety of places to discover, with a different environment surrounding it. The island is also home to plenty of hostile creatures that attack the player on sight.

The game follows a player that wanders this island. The island is always different due to procedural generation, however, the idea is to have the island follow a specific pattern that can influence the player's knowledge during another playthrough. The main idea is to have multiple distinguishable biomes scattered around the island.

The player has to overcome obstacles to achieve the goal. Obstacles in form of the enemies that try to eliminate the player at any cost. Multiple biomes do offer multiple varieties of enemies wandering on the island. Enemies with different sprites, sounds, and difficulty to beat are the key element in the game. Enemies design is adapted to a procedurally generated world with obstacle avoidance and a pathfinding algorithm.

With playable character and enemies, a proper combat system is extremely important. The combat system is only in the melee variant, with a simple targeting system to deal with multiple enemies. The player also disposes of multiple usable abilities to defeat foes. Combat system uses health system to determine the state of a character, whether it is alive or dead. An RPG game's signature system is that numbers pop up when dealing damage or healing to enhance the experience. This system closely relates to health and combat systems. The game also saves the progress of the player.

Last but not least UI experience is an important part of the game. A simple starting menu to start a new game or load previously played is extremely important. Save system, is in some cases automatic but the player can save manually in the pause menu as well.

The game should present a classic RPG game with a 2D top-down view and a simple goal game-winning condition.

4.2 Scope definition

Defining the scope of a video game is always a difficult task. The main reason is, that it is quite easy to add more features as the game progresses through the development. However with limited time and focus on appropriate aspects, this game consists of a procedurally generated map, multiple enemies, and a combat system related to the RPG genre. Game mechanics can be divided into two sections. The first section is related to creating the world. This part explains processing Perlin Noise into a scalable and adjustable world. The second section outlines gameplay-related problems.

The map has a square structure with 512x512 dimensions. Each time player starts the game, a seed is evaluated and a map is constructed. The whole process from seemingly random numbers, all the way to the map is described in section 4.4. The map consists of an island surrounded by the ocean, where multiple biomes and enemies can be found. These biomes are *the Ashland, the Forest, the Desert, and the Rainforest.* Biomes and enemies are further described in sections 4.3.3 and 4.6.1 respectively.

Player is able to control main character. Main character has assigned several *abilities* further described in section 4.5.2, and health in section 4.6. If players health drops to zero game is over, on the contrary winning conditions are achieving four missing keys described here 4.3.

4.3 Game rules

The player is thrown into the generated world. His main objective is to collect all the missing keys to finish the game. Each key can be found within a different biome across the map. Each biome consists of exactly one key, however, there might be multiple biomes of the same type. ***Gathering all missing keys is required for the player to finish the game***, however difficulty differs by the biome itself and enemies that can be found within them. The main purpose of this design is to make game playthrough more variable as the player explores the world.

4.3.1 Controls

The gameplay consists of the character's movement, attacks, and ability usage during combat. Character's movement across the world can be controlled by **WASD** input keys. Combat consists of *regular attack* that can be performed by **left mouse click** near target's position. Player is given *five abilities.* These abilites and assigned keys are:

Dash *(SPACE)* - quick movement burst in running direction.

Shied *(RIGH MOUSE)* - absorption of damage

Heal *(Q)* - heals character after casting of the spell is done

Tornado invocation *(F)* - instantly casts 8 tornadoes hurting everything in its direction.

Sword clash *(E)* - AOE[1] spell that spawns swords on top of enemies within range and damages them.

Each ability is further described in section 4.5.2 below. Camera movement is stuck to the player's position, meaning the player is always in the center of its view.

4.3.2 The main quest

As mentioned above, the goal is to find four missing keys that can be found within each biome. These keys are hidden within *the spheres*. Spheres can be interacted with, and after a short period of interaction, a key is acquired. Interacting with all four spheres is required, afterward winning screen is shown and the game is completed. Spheres are guarded by the *Cultist* enemies. After acquiring the key, *guards* no longer spawns, and the sphere remains broken. Also, the lock icon in HUD is unlocked and the arrow pointing towards the sphere is gone as shown in Figure 4.1. Enemies and their stats can be inspected in the table 4.1 below.

name	**health**	**damage**	**movement speed**	**attack time**	**attack range**
Undead	185	16 - 21	1.4	2	2
Necromancer	125	20 - 26	1.1	2	7
Skeleton	200	9 - 14	1.6	1	5
Goblin Beast	250	25 - 35	1	3	5
Goblin	125	5 - 12	2	1.2	5
Cultist	125	20 - 40	0.7	3.5	7

Table 4.1: Shows enemies and their basic attributes. Health makes enemies more durable. Higher attack time means, more time is required between each attack. Attack range defines how far attack can reach before missing.

(a) HUD with four *incomplete* keys.

(b) Player approaching found keystone.

Figure 4.1: Shows Heads-up display visible during gameplay. Figure 4.1a indicates that there are 4 keys to be found, and arrows above them acts as a guide. Figure 4.1b on the contrary shows that player has collected all missing keys, meaning the game is complete.

[1] Area of effect - is a spell type found in RPG games that affect a wide area on the ground where everything within that area takes damage/healing

4.3.3 World

Procedurally generated map has dimensions 512x512. Map is constructed of tiles[2] that differs by the biome and position. Some tiles can be walked on while others can not. Player can encounter *six* different biomes.

- **The Ocean** - located around the island.
- **The Ashland** - located on the poles of the island.
- **The Forest** - region between poles and hot locations near equator.
- **The Rainforest** - hot, moisture biome located near center of the map
- **The Desert** - hot, dry biome located near center of the map alongside *The Rainforest*.
- **The Beach** - conects ocean and other terrestrial biomes.

Each biome consists of different vegetation and enemies and offers a slightly different experience. Biomes worth noting are mentioned below.

The Ashland

Biome that is located in the very north and south of the island. Player usually spawns within this biome. The Ashland is dead land full of dried plants and bones. Vegetation does not exist, only a few trees can be found within this biome. Enemies wandering these lands are the *Undead*. They are average paced and their attacks are easy to dodge.

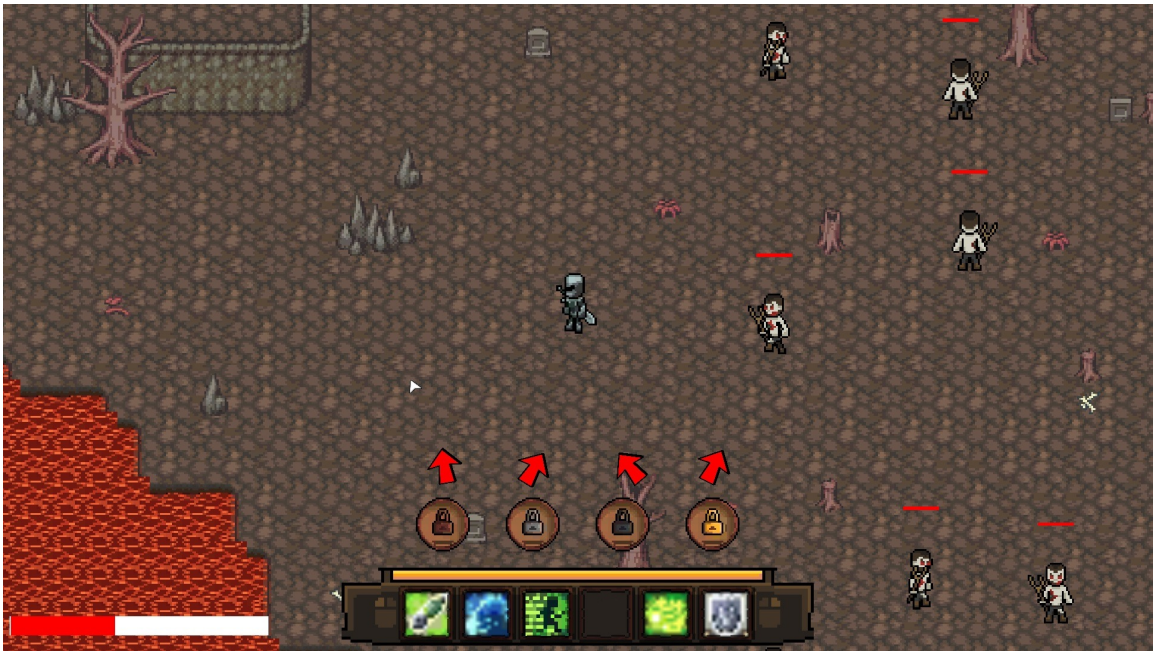


Figure 4.2: Player in *The Ashland*. Bottom left corner shows water(lava) type related to this biome. In the right side are spawned *undead* enemies.

[2]The smallest portion of the map

The Forest

Usually located between The Ashland and The Rainforest of the Desert. This biome consists of a great number of vegetation such as flowers, grasses, and trees. Tree occurrence is common and quite dense. *Skeletons* are player's enemies in this biome. They are faster than the *Undead*, however, deals less damage but are more durable.

Figure 4.3: Player moving around The Forest, while being noticed by skeletons that are ready to attack.

The Desert

The occurrence of this biome is related to high-temperature locations. This means that this biome is usually located in the center of the island. Vegetation is similar to *The Ashland* biome, however, the enemy found within this biome is called the *Necromancer*. Necromancer's attacks are hard to dodge due to the big attack range.

The Rainforest

Its occurrence is also in hot regions, meaning near the center of the island. It usually borders The Desert and The Forest. Vegetation is very dense, especially trees. Two types of monsters can be found within this biome. *Goblin* is very fast but deals little damage while having low health points. The second type of monster is the *Goblin Beast*. *Goblin Beast* (bottom right of the 4.5 figure) is a huge monster that is very slow-paced. Also, his attacks are slow but deal much bigger damage. It is the most durable monster in the game.

Figure 4.4: The Desert biome and its emptiness. Usually located near the Rainforest. These two biomes are exactly the opposite. While the rainforest offers a huge variety of vegetation, the desert is extremely hot and devastated land scattered only with rocks and sand.

Figure 4.5: Player in The Rainforest moving throught the dense forest. In the bottom right is one of the enemies *Goblin Beast*. Navigate through this biome is not an easy task, and requires to be alert all the time.

4.4 World generation

First of all, there must be a world to work with, for the character to walk on and enemies to spawn within. Generating terrain requires putting random values into the heightmap as mentioned in chapter 3.3.6. Generating all the values directly into the heightmap is not the best idea, since the result will look too noisy.

The map is a 2D array divided into chunks and tiles. One chunk consists of 256 tiles. Each tile stores values of height, moisture, and heat. These values are generated by the Perlin Noise function. As mentioned, three noise maps are generated to achieve desired world generation. Each of these maps represents different attributes of the world.

Heightmap generator attributes are frequency and exponential of the noise. The higher the frequency, the further apart the sample points will be, which means the values will change more rapidly. Increasing the exponential of the noise will result in samples being more aggressive. The terrain was designed to always generate island structure, meaning the edges of the map are naturally bordered by the ocean as shown in Figure 4.6. The altitude of the tiles could potentially affect temperature, meaning overall biome selection, however, this functionality did not provide sufficient results.

4.4.1 Biome generation

Generating biomes was achieved using precipitation and temperature noise maps, and their subsequent application in the Whittaker diagram. The precipitation map represents average rainfall. Its attributes are persistence and lacunarity. Distributing precipitation and heat properly was a difficult task to achieve generation in sufficient form.

4.4.2 Objects and vegetation

Objects and *trees* are generated in a different way. *Trees* are generated using their own Perlin Noise map. This map is subsequently processed, where in the specified radius (radius differs for each biome) the highest perlin value is selected and that position is marked as a tree. Trees do not require to be saved, because of the usage of Perlin Noise. Noise is generated using world seed with offset, meaning trees are always placed in the same positions. Their assigned sprites, however, have to be stored in JSON files.

Objects, on the other hand, are placed within each chunk completely randomly. An array of tiles within the chunk is shuffled with the Fisher-Yates shuffle algorithm, and the maximum number of objects is consequently placed. Objects require to be saved in a JSON file for each chunk. The position of the object and its according sprite are saved for every single object spawned within the world. A maximum number of objects for a chunk is fixed to 20. Objects and trees pooled from the Object pool design pattern.

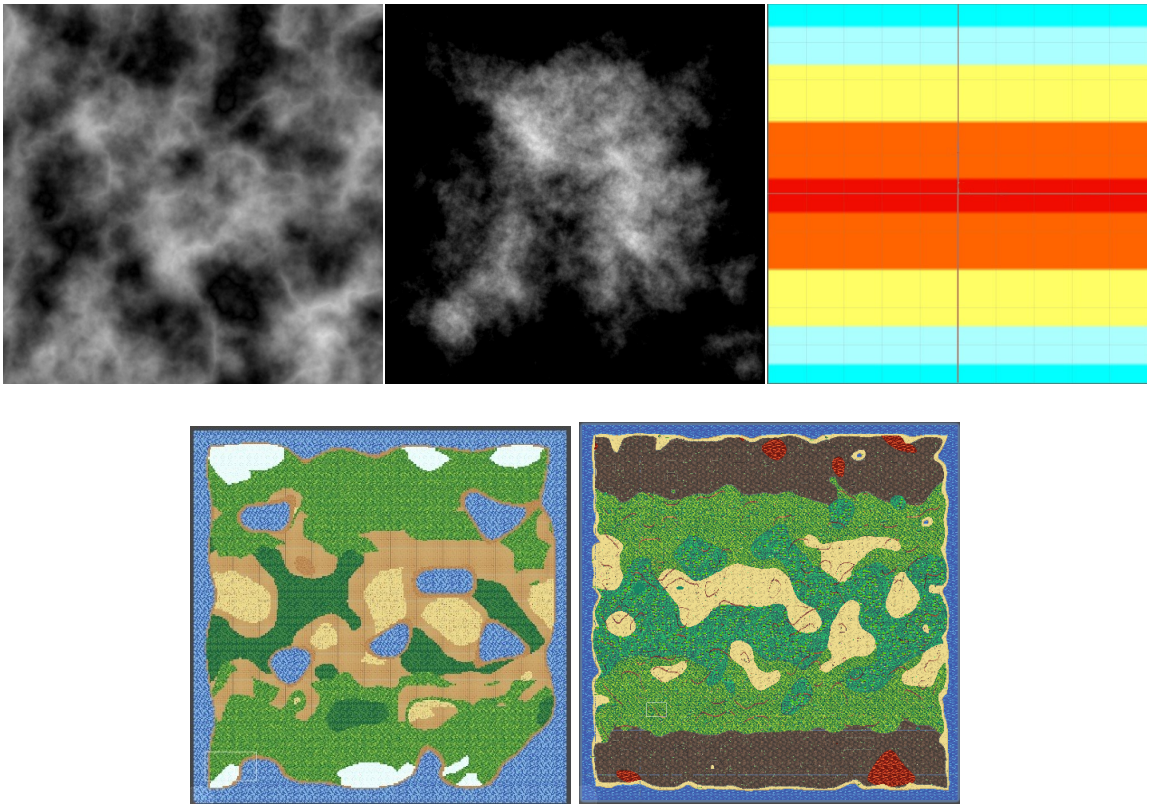


Figure 4.6: First row shows three crucial noise maps, moisture map representing average rainfall, height map stands for altitude and heat map showing average temperature (respectively). These are further merged and processed to form biomes. Second row from the left shows world generation in early stages, while right figure shows properly generated world with hills and cliffs.

4.5 Characters

Character sprites are basic and are 48x48 rectangular images. Each character consists of animations, such as walking, attacking, and so forth. These animations work in four directions. Up, down, left and right and are dependent on the current situation within the game.

4.5.1 Movement

Each frame must be triggered in different stages to perform the animation or idle state. Walking animations are dependent on input keys, therefore no other calculations are required. On the contrary, while dealing with mouse-related animations, the angle between the horizontal vector and the mouse position vector is calculated to determine the proper direction of animation to be played. This system is described on the figures 4.8 below.

The player's anchor point is handy during this calculation. The horizontal vector moving through the anchor is compared with the position of the mouse. As shown in Figure4.8. This action is performed every frame within the Animation Controller script.

(a) Walking down animation used in the game consists of eight frames

(b) Attacking animation towards left side

(c) Kneeling animation in upwards direction is used in *shield* ability.

Figure 4.7: Figures shows few of many animation sprites used throughout the game. Sprites are in PNG format, and are rendered by *sprite renderer* component in Unity. Animations are done by Unity's *Animator* component with appropriate controller.

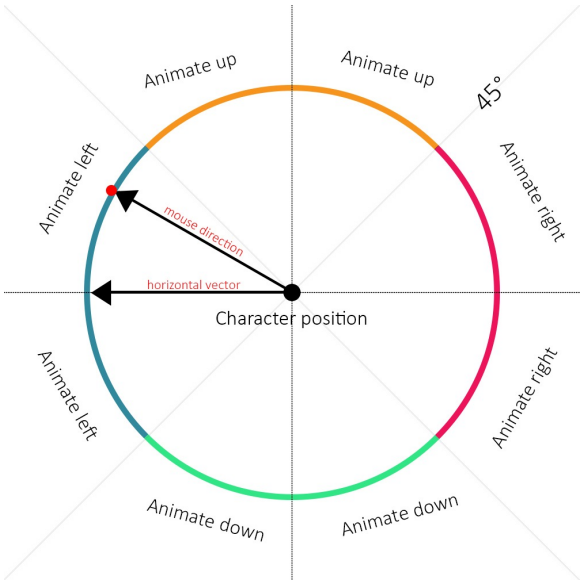


Figure 4.8: Calculating angle between horizontal vector, and vector from character position towards position of the mouse. This action is performed every on frame for each character present in the world.

4.5.2 Skills and abilities

Player has 6 *active*[3] abilities, and one *passive*[4] ability to work with. Active abilities can be divided into defensive, offensive, and utility, each usable in different scenarios offering different effects on gameplay. Abilities are closely bounded with the HUD system described

[3] Active ability requires player's interaction and usually requires player's precise timing and skill.

[4] Passive Abilities are passive, always-active modifiers to either game mechanics, abilities, or hero behaviors.

in section 5.7. Each spell has assigned different *cooldown time*[5]. The remaining cooldown time is shown on top of the ability icon after being cast. Abilities also work with the Popup system that is described in the section 4.6, whether it means dealing damage, healing, or absorbing damage.

Dash

Dash is a blink/movement/travel skill which causes the skill user to quickly move in the movement direction. It is considered to be a utility skill to become more mobile. Its primary function is to dodge attacks and travel faster. Dash can be performed by *SPACE* key. Has cooldown of **5** seconds.

Shield

The shield is a defensive ability that causes to absorb incoming damage from the enemies. While shielding character is unable to move. Shield has 100 hp and has no cooldown. The shield can be invoked by holding *RIGHT MOUSE BUTTON*. After the shield is broken, the player's health is damaged until the shield regenerates.

Shield regeneration

The only passive ability in the game. When shield's health points are not full and the player is not in combat, the shield ability starts to slowly regenerate. This behavior can be seen by observing the popup system while playing.

Heal

Heal ability is the only ability that requires the player to cast[6]. After cast time is done (5 seconds), the player is healed for **15 - 45** health. Heal can be cast by *Q* key. The player must wait for **20** seconds before using heal again.

Tornado invocation

Instantly invokes eight tornadoes, moving in 8 directions and hurting every enemy it touches. Tornadoes travels for **1.5** second and deals **10** damage on contact. Tornado invocation has *10* seconds cooldown. Tornado invocation can be performed by *F* key, followed by a left-click.

Sword clash

Spawns swords on top of enemies within radius range and deals 12 damage. Extremely useful when fighting multiple enemies at once. Has *8* seconds cooldown. Swords can be cast by *E* key.


(a) Dash icon


(b) Shield icon


(c) Heal icon


(d) Sword clash


(e) Invocation

[5]The minimum length of time that the player needs to wait after using an ability or item before it can be used again.

[6]Cast time - the time needed to cast a spell or ability before it takes effect

(a) Player casting *Sword clash* ability on *necromancer* enemy entity. Both are within range of the spell.

(b) Showcase of tornado invocation. Tornadoes will travel further away from player in spawned directions.

(c) Player after casting *Heal*. As popup (described in 4.6) indicates, player is being healed for 44 points, and must wait for 20 seconds as shown in the HUD.

(d) Player absorbing *necromancer's* attack by using *shield* ability. Popup shows that damage is being absorbed. Shield health is show in the bottom right.

Figure 4.10: Showcase of *four* abilities throughout gameplay, each used in different situation and biome.

4.6 Combat and gameplay

The gameplay experience is the most important part of the game. The biggest part of the gameplay is to encounter enemies and defeat them. The main parts of the combat system are health system and multiple active and passive abilities required to use in order to progress. Targeting system, abilities, and enemy design are closely described below.

4.6.1 Enemy design

While playing the game, player has to overcome some obstacles to keep entertainment. For this purpose enemies were designed. Enemy entities do spawn on every loaded chunk, however, their spawn number is random in a predefined range. There might be *zero* enemies within the loaded chunk. Enemy entities are designed in melee[7] combat only. Each enemy has a predefined range of attack damage, health points, movement speed, attack speed, and attack range. Each enemy is also assigned to spawn in a specific biome. *Special* kind of enemy is *cultist*. This entity only spawns in swarms near the *keystones* to defend it.

[7]Any combat that involves directly striking an opponent at ranges generally less than a meter, especially using martial arts or melee weapons

Behavior of the enemy is based on random values. After spawning, enemy entity starts to wander in various directions, but still remains near its spawn point. If player's character comes closer, pathfinding algorithm calculates the closest path from the entity's position to the player's position and it starts to follow. If an enemy is close enough to the player (distance differs by the actual enemy) pathfinding stops and the entity starts to attack. If for some reason, the player is no longer within *aggro radius*[8], the enemy will follow until the last known position.

Health and combat system

To be able to actually implement a combat system, the health system has to be present. Health is an attribute that determines the maximum amount of damage that a character can take before dying. Each entity has a health bar attached to it to be able to perform combat actions. The Health system and targeting system are closer described in chapter 5.2. If entities health drops to 0 it is considered to be dead and no longer a threat.

Targeting system requires a direction vector from the player towards the mouse position to be calculated. Afterward, attack animation is played. To handle multiple enemies near the player, the relation between the closest enemy to the player and the mouse position is used. Working example is shown in Figure 4.11.

Figure 4.11: Player performing basic attack on entity. Health bar is visible above entity's head, and popup system shows actual damage amount dealt to the entity.

[8]The distance at which the entity will attack

4.7 Menu and HUD

The menu and heads-up display are the two most important parts of the UI[9] in the game. It is also responsible for navigating the player through multiple scenes and saved games. The player is also able to stop the game while playing with the *ESCAPE* button, leading to the pause menu.

After launching the game, the first thing that the player will encounter is the main menu scene. This menu consists of three buttons to play a game, load a game and quit a game. The next thing on the screen is the seed slider. This slider determines the seed of the world when selecting the play button. Saved games are located underneath the load button. This scene includes every saved game divided by the seed number. Custom names for the saves are not supported, meaning the player has to remember the seed number in order to load the correct game. The main menu can be seen in Figure 2.6 in the overview chapter.

The heads-up display(HUD), is the most noticeable part of the UI during gameplay. HUD consists of the player's health bar. Shield's health bar is visible during activation of the shield. The biggest part of the HUD is the action bar. The action bar consists of usable abilities and their responsive cooldowns. After the usage of any of the usable abilities, the ability icon is grayed out, and the cooldown number in seconds is displayed instead. If an ability is ready once again, the player will be alerted with the corresponding sound, and the icon within the HUD is grayed out no more. HUD, alongside other things, is visible in Figure 4.12.

Figure 4.12: Shows a screenshot taken from the game. The figure contains descriptions of individual parts of the UI and the game itself. Figure is in the exploded-view format describing the scene composition and relation between the UI and the game. Player's health bar, action bar, objective guidance and shield's health bar are considered as HUD.

[9]User Interface

Chapter 5

Implementation

This chapter contains information about the implementation of the game, as the development progressed multiple problems to solve naturally occurred. The game was implemented in Unity Engine version 2021.2.13.f1, which offers a choice between UnityScript and C# languages. C# language was chosen for its huge variability and integration, and huge support throughout the developers‘ community. Also, Unity announced they are deprecating UnityScript, and will no longer be supported.

As mentioned the whole game was implemented in C# language and was handwritten. No external script assets were used and the main source of information was Unity's documentation. Visual Studio Code was primary IDE[1] for its great integration with C# language and Unity Engine.

The game itself and its components were solved in multiple stages. These stages are chronologically mentioned in this chapter as they were implemented. Starting with the implementation of the world and its optimization, followed by the implementation of the player's movement around the world and basic attacks. Afterward designing the enemies and implementing the health system followed by HUD[2] and menus. Lastly, implementation of sounds and visual effects was made to enhance the gameplay experience. Rough structure of the game is shown in Figure 5.1.

[1]Integrated development environment is software for building applications that combines common developer tools into a single GUI.

[2]Heads up display - details of the player visible in the screen

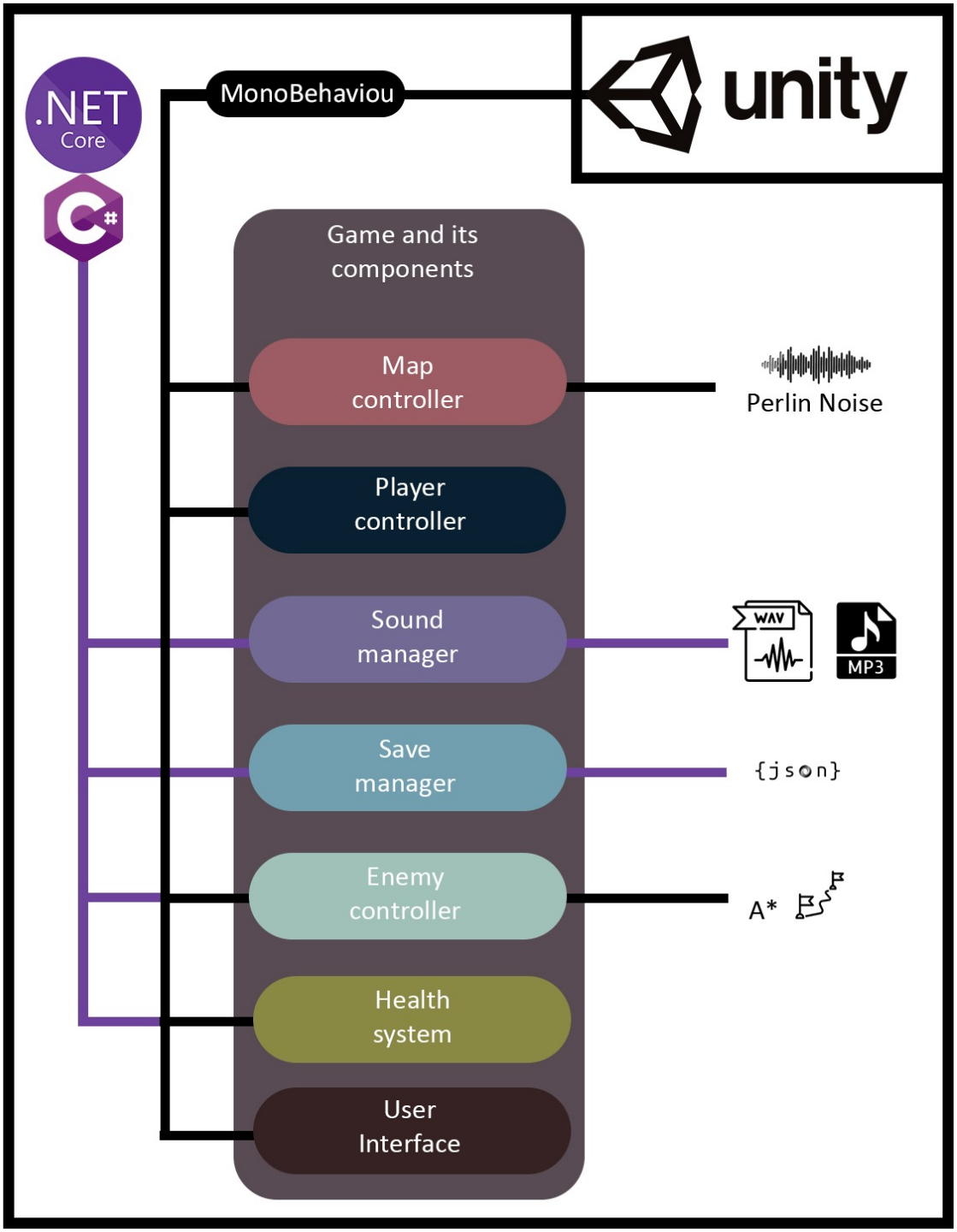


Figure 5.1: Shows game and its components in block scheme. Figure also shows each component's main implementation source. Some components are implemented purely by using **C#** language, such as *Sound manager* and *Save manager*. In particular *Save manager* benefits from System's JSON serialization. Others derivated directly from *MonoBehaviour* [3] and are attached to GameObjects. *Health system* and *Enemy controller* on the other hand, utilizes both *System libraries* as well as *MonoBehaviour* provided by Unity Engine.

5.1 Map controller

As mentioned several times, map is procedurally generated with the usage of Perlin Noise, which is further processed and modified to match the desired output. The detailed process of the whole generation is described below. Perlin Noise is implemented in Unity's *Mathf* library.

Processing the noise

Beginning in the **ChunkGenerator** class, three Perlin noise maps are generated, one representing *height*, while other two representing *moisture* and *temperature* of the world. Each map is generated with different parameters but with the same world seed to generate non-identical results.

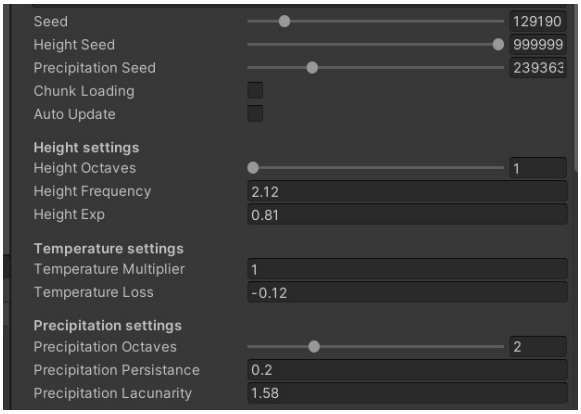
Height map is further processed in **MakeIslandMask** method (algorithm can be found here: 1), which is responsible for cutting out corners of the map, while *moisture* map is populated ordinarily. After this process, height noise values on the edges of the map are approximating zero, resulting in generating a square-like island as shown previously in Figure 4.6. *Temperature* map is constructed in gradient structure. The highest temperature is near the center while approaching the north and south edges of the map, temperatures drop to cold values. This transition is done by simple calculation 5.1 and should imitate the earth's temperature conditions.

$$heat = \frac{latitude}{\frac{map\_height}{2}} * temp_multiplier - \frac{elevation}{temp\_loss} \tag{5.1}$$

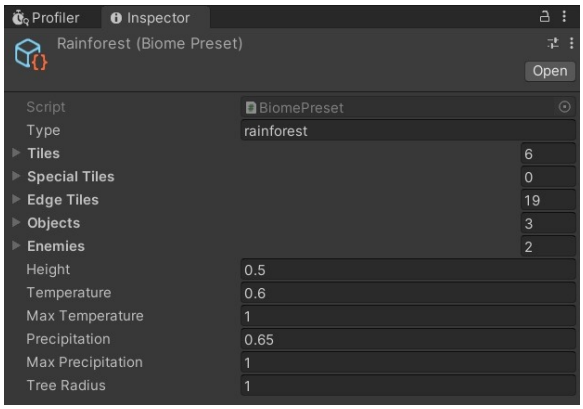
$d \leftarrow GetDistanceToEdge(x, y)$;
if $i < minIsland$ **then**
  $landmass \leftarrow false$;
  return 0; `/* outside of desired island, return height 0 */`
else
  if $d \geq maxIsland$ **then**
    $landmass \leftarrow true$;
    return $oldValue$; `/* return generated height */`
  else
    $factor \leftarrow GetFactor(x, y)$;
    if $(factor \times oldValue) < threshold$ **then**
      $landmass \leftarrow true$;
    else
      $landmass \leftarrow false$;
    end
    return $oldValue \times factor$; `/* return if tile is solid or not */`
  end
end

Algorithm 1: MakeIslandMask method for island structure (simplified). This method is called for every single tile on the map.

Generated heat, height, and moisture maps are subsequently merged. By merging their values, each tile can be assigned a biome to distinguish diverse parts of the map. For that purpose, *BiomePreset* 5.2 was designed. Results are shown in Figure 5.3a.



(a) Perlin noise attributes for world generation.



(b) BiomePreset class in Unity's inspector.

Figure 5.2: Shows Unity's interface with two different windows. Figure 5.2a is script component attached to the **Map** GameObject that contains attributes for ***ChunkGenerator's*** generating method. Each attribute affects generation in different manner. For example increasing *frequency* of the height, will result in more hilly terrain. Figure 5.2b shows preset instance of the class editable in the inspector panel. Each biome is based on this class. Tiles that meets these requirements, will be evaluated as *Rainforest* biome.

Evaluating noise into tiles and map

TDMap class represents map structure, while *tile* is the smallest portion of the map that is represented by ***TDTile*** class. According to specified criteria of the biome (such as these for the *Rainforesh* 5.2b), each tile's *biome* is evaluated. Rules for determining corresponding biome are minimum and maximum values of height, moisture and temperature as shown in Figure 5.2b. Decision of which biome should be assigned to each tile, is done by ***EuclideanDistance*** method within *BiomePreset* class. This method simply uses 5.2 formula, for calculation of Euclidean distance between generated values of the tile and each biome's preset values. Shortest distance is then assigned as final biome. Results are quite satisfying and can be seen in Figure 5.3a.

$$d = \sqrt{(noiseHeat - biomeAvgHeat)^2 + (noiseMoisture - biomeAvgMoisture)^2} \quad (5.2)$$

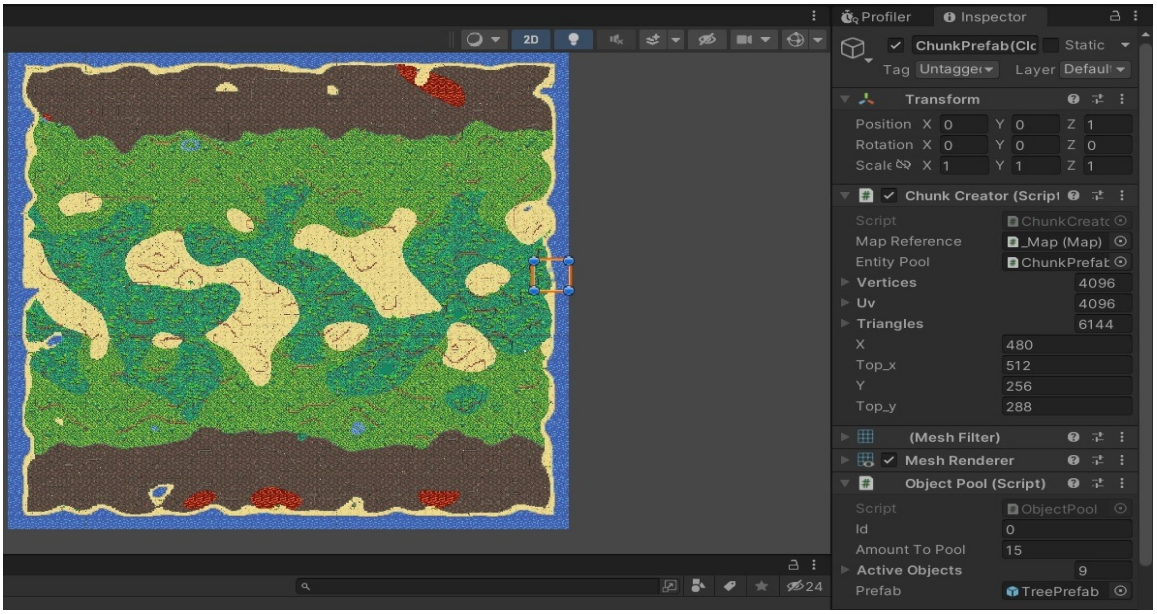
Optimization

The current map size is fixed to 512x512 tiles, however, this number could be potentially infinite. Rendering the whole map is also a waste of resources, and generating that many GameObjects for every single tile would be a huge performance drawback, that is why *chunks* and *object pools* were implemented.

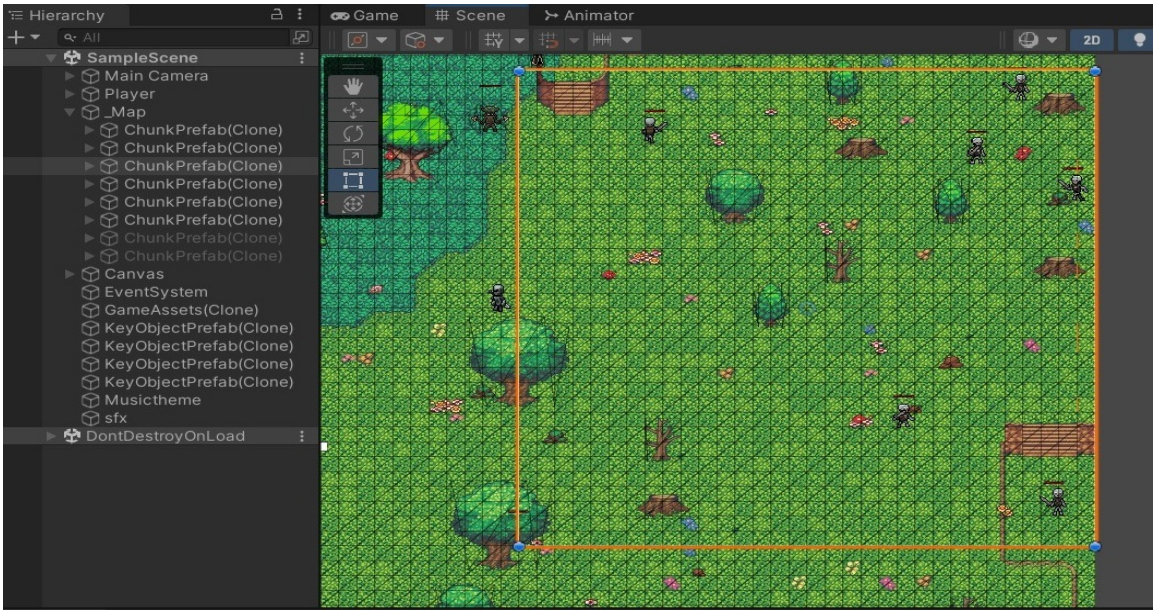
One chunk is represented by ***WorldChunk*** class that hold crucial information about chunk itself. It has attached mesh component, that is divided into 32x32 quads. Chunk GameObjects are called **ChunkPrefab** with ***ChunkCreator*** script component attached to it as shown in Figure 5.3a. This script is responsible for dividing and texturing the mesh. Specifically *CreateTileMesh* method is responsible for calculating **uv** coordinates for every single tile and set proper textures for each quad. This resulted in huge performance boost when creating *single* GameObject whereas before *1024* were required. Single *ChunkPrefab* GameObject, and its mesh divided into quads can be seen in Figure 5.3b.

Another performance boost was achieved by implementing the *object pool* interface. Instead of constantly instantiating and destroying GameObjects, a pool of objects is created in the beginning, and periodically are being loaded and unloaded into the scene. This system is also extremely useful later on when dealing with entities and vegetation.

With player moving around the world, chunks and their contents are regularly being loaded and unloaded depending upon the player's position.



(a)



(b)

Figure 5.3: Figure 5.3b presents fully rendered map with chunk loading turned off. Map consists of 256 chunks. One chunk GameObject is highlighted, while on the right side in *inspector window* are its attached components. In Figure 5.3b single *ChunkPrefab* GameObject is rendered in wired view. Individual quads of the mesh are visible.

5.2 Health system

Health system is fundamental part of the combat. System is built in two part. First is *HealthSystem* class that is defines basic behaviour for the system as shown in structure 2.

Class *HealthSystem* **contains**

```
// notifies observers about health being changed
event EventHandler<ShieldEventArgs> OnHealthChanged;
int health;
int healthMax;
void HealthSystem(int maxHealth);
void SetHealth(int amount);
int GetHealth();
void Damage(int damageAmount);
void Heal(int healAmount);
void HealMax();
float GetHealthPercent();
```

end

Algorithm 2: Health system class (simplified). Attributes and methods are self-explanatory. *OnHealthChanged* notifies all subscribers that the health of this instance changed. Subscriber is *HealthBar* attached to entity.

Second part consists of *HealthBar* GameObject. This GameObject has multiple child GameObjects to perform tasks required as shown in Figure 5.4. Its only job is to show the player current health status of each entity. Every health-related tasks are performed only by the *HealthSystem*, which relies on C#'s **events** to notify *HealthBar* when change to health occurs. *ParticleSystem* component is also attached to enhance visuals when the entity is damaged by imitating blood.

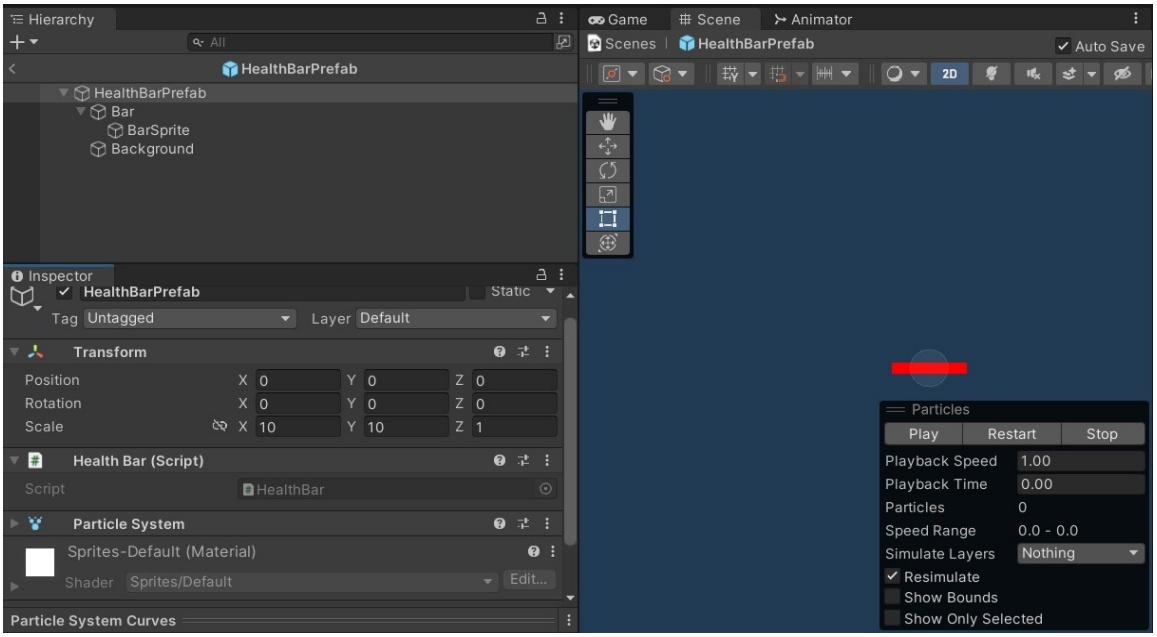


Figure 5.4: *HealthBar* GameObject's structure in Unity Inspector. This GameObject is instantiated alongside with each entity(player included). HealthBar and *ParticleSystem* components are also attached as show in the bottom left.

5.3 Player Controller

Player controller consists of multiple components as shown in Figure 5.7. Each handles different problems encountered during the development stage.

The Player has attached the *Rigidbody 2D* component which is responsible for physics related to 2D sprites. This component is used mostly for all movement-related work. Movement is done simply by Unity's input listeners in *Update()* function meaning for each frame input key is checked, whether it is pressed down or not. Physics-related work is done in frame-rate independent *FixedUpdate()* method designed for physics calculations. Simple movement is done by:

$$rigidbody2d.velocity = moveDir * movementSpeed;$$

Animation Controller is responsible for all animations related to characters. Fundamental part is system described previously in chapter 4.8. One of Player GameObject's children is *AggroCollider*. It is responsible for enemies within range to attack the player. Method **OnTriggerEnter2D()** is attached to CircleCollider2D with specified radius. If entity triggers this event, *observer* pattern is used and entity subscribes to player. This solution is performance-wise extremely effective.

(a) Enemy is out of collider. (b) Enemy within collider. (c) Enemy attacking.

Figure 5.5: Shows functionality of *AggroCollider* component. In Figure 5.5a enemy is not interested in player, since it is out of radius. On the other hand, in Figure 5.5b enemy has triggered OnTriggerEnter2D and started A* pathfinding towards player. Figure 5.5c shows that enemy has reached player and starts to attack.

When facing multiple enemies around player, targetting correct one is important. This is achieved by using *GetClosestPosition(attackPosition, attackRange)* method within **EnemyController** script which is basic sequential search.

Abilities

Dash ability works with movement. Main implementation is in **Dash()** method in the PlayerController script (pseudo-code shown in 3). During dash ability player becomes invincible. This is achieved by **BecomeTemporarilyInvincible()** coroutine, which lasts for dashes duration. If the player is attacked during this time period, a miss will occur instead of damage dealt. Ability is also controlled in terms of irregular dashes. Such as

```
Method Dash is
    StartCoroutine(BecomeTemporarilyInvincible());
    dashPosition ← playerPos + dir * dashAmount;
    if CheckLandingPosition(dashPosition) == false then
        SoundManager.PlaySound(Sound.Error, playerPos);;
        return
    else
        raycast ← Physics2D.Raycast(playerPos, dir, dashAmount);
        if raycast ≠ null & uiHandler.DashCooldown() then
            rigidbody2d.MovePosition(raycast.point);
            SoundManager.PlaySound(Sound.Dash, playerPos);
            rigidbody2d.velocity ← dashDir * dashAmount;
        end
        SoundManager.PlaySound(Sound.Error, playerPos);;
        return
    end
end
```

Algorithm 3: Dash method (simplified). Thes method is called after *SPACEBAR* is pressed. If dash can not be performed because of wrong landing position or cooldown, *SoundManager* notifies the player by error sound. Dash itself is executed by adding velocity.

dashing into the obstacles or over the cliff. **RayCast2D** is used to determine dash landing location, so if raycast hits a collider, the position is set into collision position.

Shield attaches another *HealthBar* instance to the player. While shield is active, its HealthSystem is taking damage instead of the player's main health. Shield's visuals are achieved by using Unity's *Particle system* and *Shader Graph*. A particle system is bound on *RIGHT MOUSE* hold to start emitting emissions, creating shield's visual effect. Also, shield's HealthBar is visible in HUD during the shield's activate period.

Tornado Invocation ability is simple ability with 2 stages. After pressing bound hotkey[4], the script enters the first stage that acts as guidance. This stage uses Unity's *Line Renderer* component to expose trajectories of tornadoes about to spawn. After using *left mouse click* in this stage, tornadoes will invoke and start moving in their predefined patterns.

Sword Clash is also a simple offensive AOE ability. Main part of functionality is handled by the **GetEntitiesWithinRadius()** method, which after passing position and radius returns list of entities. Afterwards, each entity is damaged, followed by instantiating sword GameObjects on top of each entity.

[4] a key or a combination of keys providing quick access to a particular function within a computer program.

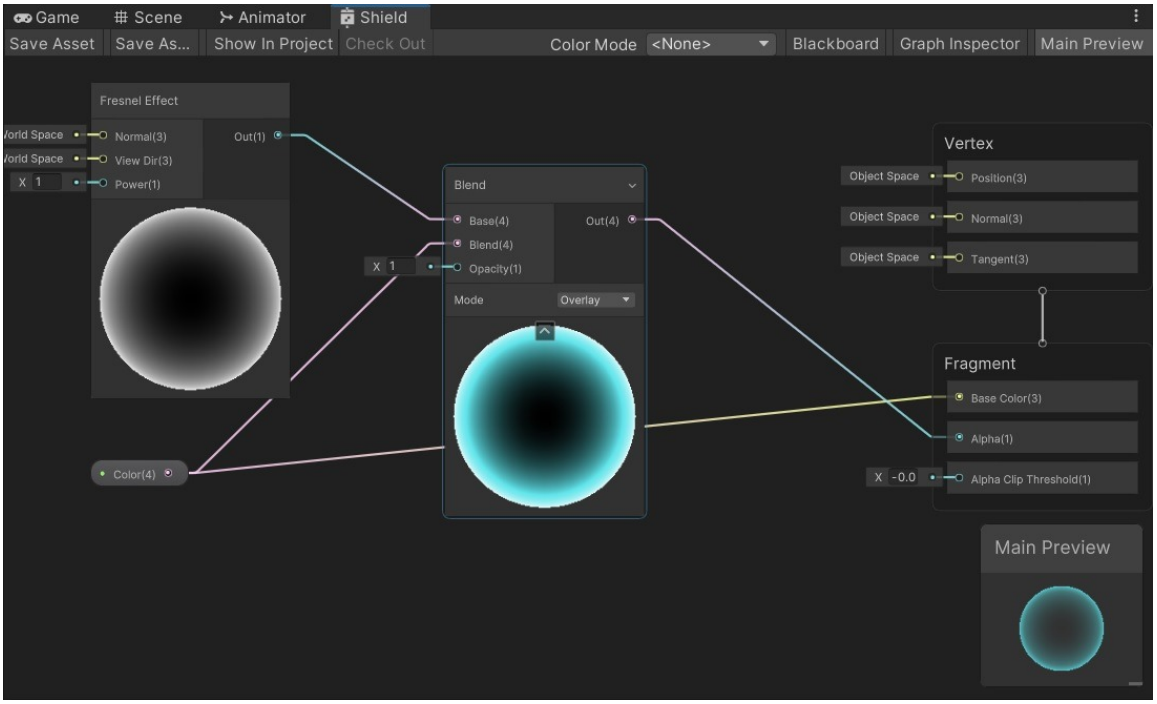


Figure 5.6: Shield's *Shader Graph* attached to *Particle System* component in shield GameObject. Shader Graph acts as particle as its emmision creates desired visual effect. *Fresnal effect* on the left, is the effect of differing reflectance on a surface depending on viewing angle. *Blend node* in the center blends two normal maps defined by inputs together, normalizing the result to create a valid normal map. In this case Fresnal effect is base and defined color is second input resulting in final bluish bubble effect.

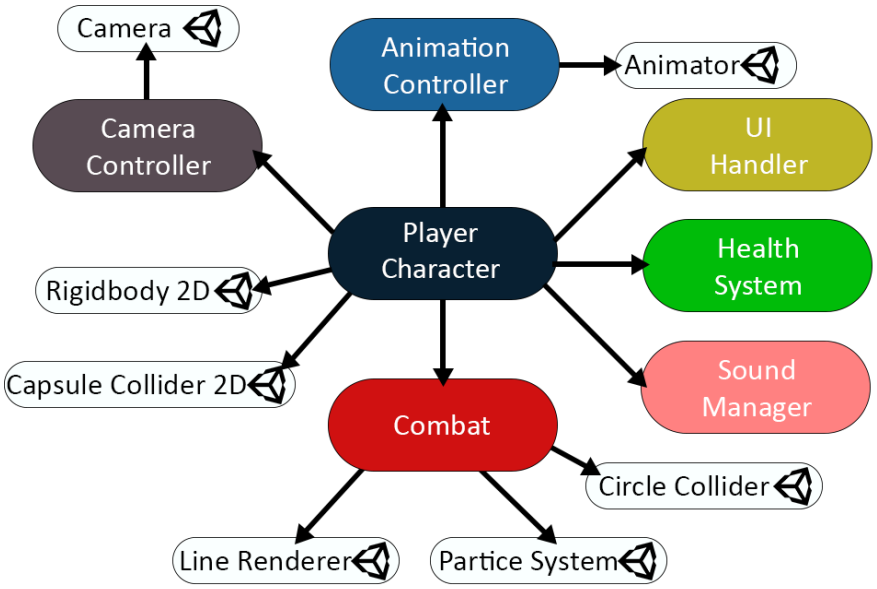
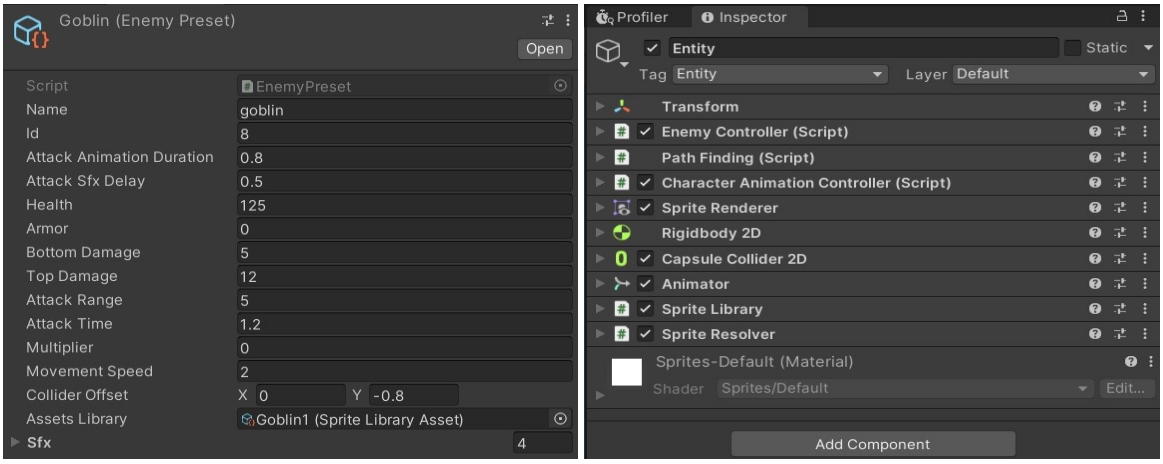


Figure 5.7: Shows player and all related components attached. *Animation Controller* is responsible for animating the sprites in various situations. *Camera Controller* is supposed to makek sure that camera always follow player. Combat mostly consist of usage of *Health-System* and implementation of abilities.

5.4 Enemy Controller

Enemy has similar components and attributes as player. Entity GameObject consists of *Rigidbody2D* to simulate physics and *CapsuleCollider2D* to specify entity's body size. Animations are handled by *AnimationController* script.

EntityPreset is ScriptableObject[5](as well as BiomePreset), holding data about each individual enemy. Enemies are instantiated in *ObjectPools* within each chunk. When activating chunk and its contents (enemies included), Unity's **OnEnable()** function is called. OnEnable is used to instantiate entity's HealthBar and set proper data from *EntityPreset*. EntityPreset's contents are shown in Figure 5.8a.



(a) EntityPreset example in Unity's inspector. (b) Entity GameObject's attached components.

Figure 5.8: Shows two separate windows in inspector. Figure 5.8a shows ScriptableObject defined for goblin entity that is further used in EnemyController script to properly set attributes for every enemy type. Figure 5.8b shows all necessary components for enemy to work.

Enemy behavior

InAggroRadius is an important **event** related to *AggroCollider* mentioned above. Enemies are spawned on top of the random spawnable tile within the chunk. Behaviour is based on a simple cycle. If an entity is **not** *InAggroRadius*, a random tile near its spawn point is selected for the entity to wander towards. After the entity reaches the selected tile, a random quantum of time is selected for entity to observe. This cycle repeats until the entity is not in an aggressive state. If entity is *InAggroRadius*, **A*** pathfinding algorithm starts navigating towards player's position (shown in Figure 5.5). New path is being calculated continuously throughout *InAggroRadius* is subscribed. The enemy tries to avoid obstacles such as trees or stones along the way, with simple obstacle avoidance detection using raycasts. This system could do much better, but time shortage required it to be implemented in a very simple and not the most effective way. After **InAggroRadius** unsubscribes from the player, the latest known position is reached, and the entity starts to wander once again. **A*** pathfinding is optimized with a heap data structure.

[5]Data container used to save large amounts of data, independent of class instances.

5.5 Sound and visual effects

SoundManager class is responsible for managing all the sounds in the game. Basic sounds are played by the **PlaySound()** method which basically instantiates the GameObject and uses PlayOneShot() function attached to the AudioSource component. Simple sound effect such as attack sound or hit sound shares 1 GameObject since it is not necessary to spawn each sound individually.

More complex sounds, such as walking requires a sound collision check. **CanPlaySound()** method does exactly that. For a given delay, *SoundManager* checks, whether the given sound was played within the specified offset or not. *SoundManager* also handles looping the main music. Looping short simple sounds, such as casting heal, was implemented using coroutine. *SoundManager* is also capable of playing 3D sound when provided the position of the object. When the game launches, the main menu theme starts to loop, as soon as a specific map is loaded the main theme is continuously looped instead.

DamagePopup

To enhance HealthSystem's feedback to player, *DamagePopup* system was implemented. It is a simple GameObject with text mesh attached to it indicating damage or healing done to the entity. This system is visible in most combat-related figures shown previously such as 4.11. DamagePopup system instantiates GameObject, alters the text mesh component attached to it with custom color and string. Afterwards GameObject has movement and disappear effect implemented in *DamagePopup* script.

Unity's Particle system was heavily used in terms of visual effects. Particles occur during dealing damage, usage of the *shield* and *dash* abilities, and in the main menu scene.

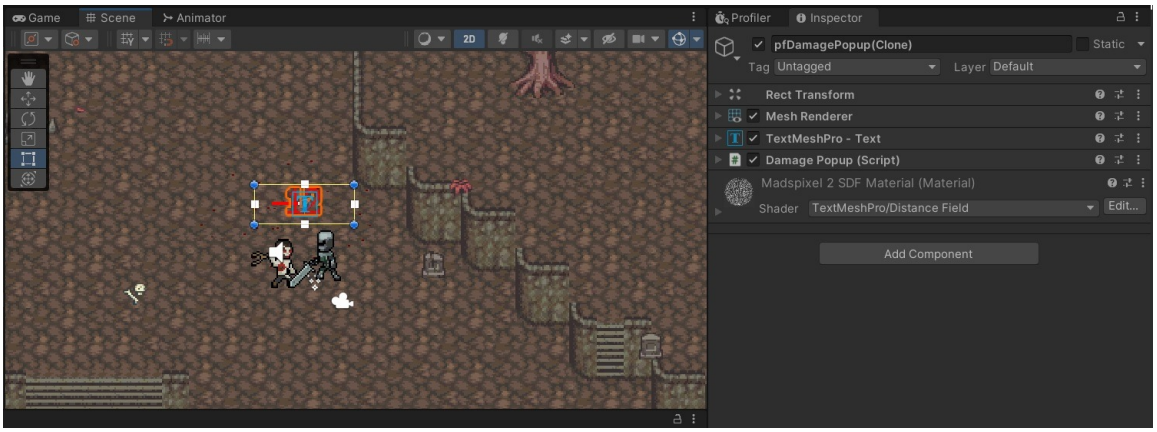


Figure 5.9: Shows DamagePopup GameObject being instantiated when dealing damage to the *necromancer* entity. Instantiated GameObject will last for 1 second, slowly moving and eventually disappears before destroying the GameObject occurs. Little sound icon on top of the necromancer indicates, that the *SoundManager* has played hurt sound effect of the enemy.

5.6 Save System

SaveSystem class is handling IO system operations when saving and loading data. Class is written without derivating from *Monobehaviour* provided by Unity. It is core component for any reading or writing in to the memory, and consists of two main methods **Save()** and **Load()**. Writing and reading is handled using **File** class provided by *System.IO* library.

GameHandler class, on the other hand, is directly attached to Unity. This class is among other things, responsible for saving data directly from the game. Two main methods **Load()** and **Save()** are implemented as generic methods, meaning any class provided to them can be saved or loaded from the memory. Provided classes must meet serialization requirements (attributes such as **int2** provided by *Mathf* library are not serializable and can not be used).

The file format for storing and reading data is **JSON**[6]. JSON proved to be the best solution since pure byte writing caused much more limitations when serializing. Saved data are divided by the world seed number, and each chunk is saved individually.

```
Class [System.Serializable] SavePosition contains
    Vector3 position;
    int healthAmount;
    int shieldAmount;
end
```

Algorithm 4: Class for saving and loading data related directly to player character. This class is passed into the generic **Load()** or **Save()** methods in the ***GameHandler***.

```
Method T Load<T> is
    saveString ← null;
    if coords ≠ null then
        saveString ← SaveSystem.Load(object, seed, coords);
    else
        if object == Objtype.KeyObject then
            saveString ← KeyObjectHandler();
        else
            saveString ← SaveSystem.Load(object, seed);
        end
    end
    retVal ← default(T);
    if saveString ≠ null then
        retVal ← (T)JsonUtility.FromJson < T > (saveString);
    end
    return retVal;
end
```

Algorithm 5: Generic **Load** method in ***GameHandler*** class. Generic method is used for multiple types of classes (such as 4) being stored throughout the game. Method uses ***SaveSystem*** to read data from given files, then serializes to JSON from using *JsonUtility*.

[6]JSON (JavaScript Object Notation) is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute-value pairs and arrays (or other serializable values)

5.7 User interface

Most of the user interface is implemented with the usage of Unity's *UI* library. This library provides basic elements, such as canvases, buttons, and sliders. Heads-up display is closely related to ability usage. Ability and its cooldown are bound to the corresponding icon. ***UIButton*** script is responsible for handling usage feedback toward the player. If the ability is recharging, the UIButton script displays a gray overlay over the icon with a countdown. Also if the player tries to use the ability during this time period, it responds with an error sound from the ***SoundManager***. This behavior is implemented using coroutines. Another important script is ***UIHandler*** which is attached to the user interface, and bound individual icons with it. It consists of getting properties for each button's cooldown.

The heads-up display is enhanced with a guidance locks. Each lock is bound to a specific Keystone and is pointing towards it. This is implemented with simple position subtraction and usage of the inverse angle formula.

Method *GetAngleFromVectorFloat* **is**
    $dir \leftarrow dir.normalized$;
    $n \leftarrow Mathf.Atan2(dir) * Mathf.Rad2Deg$;
    $if(n < 0)n \leftarrow n + 360$;
    return n;
end

Algorithm 6: Conversion from passed vector to angle with the usage of inversion. Returns the amount of rotation from the first vector to the second vector. This vector is passed as a single argument called *dir* representing direction.

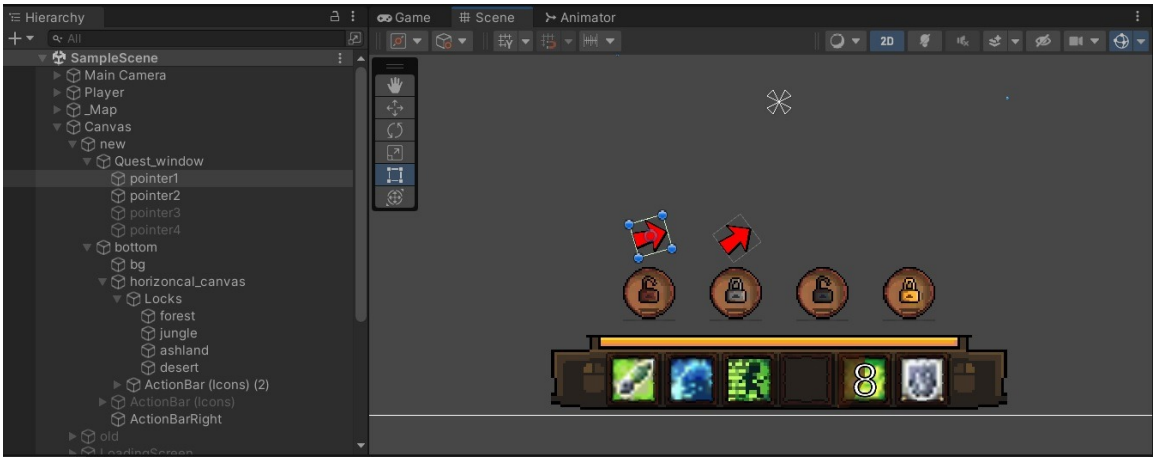


Figure 5.10: shows quest arrow pointing towards objective in the heads-up display. ***QuestUIPointer*** script is attached to every arrow to perform calculations mentioned in algorithm 6.

Menus also work on top of components provided by *Unity.UI* library. ***SceneLoader*** script is responsible for asynchronously load game scene from start menu. This is performed with *Unity.SceneManager*. **LoadSceneAsynchronously()** method is responsible for loading the scene, while indicating to user progress bar. This is achieved using coroutine in combination with AsyncOperation.

Chapter 6

Testing and evaluation

Testers were given the final product to evaluate the game quality and usability, to enhance gameplay outcomes, such as playability and enjoyment. Setting proper values for various attributes was also the goal of the testing process. Such as setting enemy health points, damage, ability cooldowns, and similar values. Also, the goal of testing was to make sure that every map seed is properly, so no impossible maps to finish will occur. Testing was done on five respondents. Each of them was given alongside the game, six questions to answer. The questions were:

Please evaluate the following

1a. Fun factor horrible □—□—□—□—□—□ fantastic

1b. Visual and music horrible □—□—□—□—□—□ fantastic

1c. Replayability horrible □—□—□—□—□—□ fantastic

2. Do you think it has potential to become fully finished game?

- □ Yes.
- □ No, the game lacks identity and is boring.
- □ Not sure.

3. What do you think would enhance the game the most?

- □ Levelling system
- □ Story
- □ Boss fights
- □ New combat possibilities (long range, magic etc.)
- □ Other: _____

4. Overall did you enjoy the game? □ absolutely □ not really because: _____

5. Difficulty easy □—□—□—□—□ hard

6. Is the game challenging to finish?

- □ Yes.
- □ No.
- □ Reasonable.

6c. How is your enjoyment of the game when losing?

horrible □—□—□—□—□ fantastic

Alongside these questions, testers have provided multiple bug occurrences that required fixing.

Reviews have shown, that the game is rather fun to play. Respondents complained about the lack of boss fights, however, the overall gameplay experience was satisfying. Each respondent showed a different enhancement preference. While 2 of them preferred boss fights, leveling system, story, and new combat possibilities also received implementation preference respectively. Visuals and music received very positive ratings. Replayability ended up right in the middle and could be enhanced much more with a variety of systems. The map although procedurally generated follows a specific predictable pattern. Reviews have shown that when a player is losing, game tends to become frustrating. The game is quite fast paced, so fast reactions are sometimes crucial for the enjoyment of the game. Several respondents claim that the game feels like a demo and has great potential to be upgraded with several concepts to fulfill it.

Chapter 7

Conclusion

The main objective of this thesis was to create a fully functional 2D procedural game. For this purpose Unity Engine was used for its huge advantages for development in small teams. Unity also provides amazing documentation and incredible community.

Game development is an enormous industry with various totally different products. Knowing this, the first step was to consolidate the idea of the final product. Procedural generation of the world was the first step of development. Various methods for a procedural content world were studied and taken into account. Implementation with the usage of noise functions and their subsequent processing was considered to be the right choice. The creation of a world that somehow meets expectations was not an easy task and took enormous time. Optimization of the world was not ideal and required to put some work. Dividing the world into chunks, and usage of the object pool design pattern was the right thing to do. However, this caused the implementation of multiple aspects to be much more complicated. This allowed for an incredible procedural generation knowledge acquisition. During this process, a study of the engine itself was a big part to understand how complicated game development actually is. After map generations proved to meet the requirements, character to walk on the map was required. Unity provides great tools to simulate physics and handle user inputs. This task was significantly easier to implement, maybe due to the time spent in the engine itself while dealing with the world generation. The next step was to design the obstacles for the player. The way that enemies are designed has a significant bearing on gameplay. Knowing this and knowing that the world has multiple unmistakable regions, enemies had to be designed in a way that the difficulty throughout the world slightly differs. That is why multiple enemies were implemented and preset for each biome. Because the world has multiple height levels, enemies had to be enhanced with a pathfinding algorithm. A* proved to be the right choice since it is widely used in the game development industry. To give the game the RPG genre feeling, the damage had to be popping out while dealing damage. This system was implemented alongside the health and combat system after designing the enemies. Experience gained before has come in handy since these systems were implemented significantly faster. Main development ended with a user interface, sound manager, and menu system followed by bug fixing. Alongside problems mentioned earlier, linear algebra-related problems were the main logical obstacles to solve.

The purpose of this thesis was to create a 2D RPG game as close to the final product as possible. Time pressure caused multiple systems to be skipped, such as leveling system, inventory, and item system, and similar. Also, a dialogue system and basic plot could be implemented and would fit the game concept perfectly. In the end, the thesis proved that

a single person is able to create a simple procedurally generated RPG game in matter of months.

Bibliography

[1] Axon, S. *Unity at 10: For better—or worse—game development has never been easier* [online]. 2016 [cit. 2022-1-27]. Available at: https://arstechnica.com/gaming/2016/09/unity-at-10-for-better-or-worse-game-development-has-never-been-easier/.

[2] Belda Calvo, L. de. *Study of procedural terrain generation in plain and spherical surfaces.* 2021. Master's thesis. LAB University of Applied Sciences & Universitat Politécnica de Valencia (UPV). Https://www.theseus.fi/handle/10024/505164?show=full.

[3] Bevins, J. *Libnoise glossary* [online]. 2003 [cit. 2022-4-23]. Available at: http://libnoise.sourceforge.net/glossary/#persistence.

[4] Bossom, A. and Dunning, B. *Video Games: An Introduction to the Industry.* 1st ed. December 2015. ISBN 9781472567116.

[5] Fandom. *World Generation.* 2022. Available at: https://dontstarve.fandom.com/wiki/World_Generation.

[6] Lisbdnet.com. *What Factors Determine Biomes?* [online]. December 20, 2021 [cit. 2022-4-21]. Available at: https://lisbdnet.com/what-factors-determine-biomes/.

[7] Meuneir, N. *Don't Starve Review* [online]. 2014 [cit. 2022-1-28]. Available at: https://www.gamespot.com/reviews/dont-starve-review/1900-6407882/.

[8] Oxford. *World Encyclopedia.* 1st ed. Philip's, 2004. ISBN 9780199546091. Available at: https://www.oxfordreference.com/view/10.1093/acref/9780199546091.001.0001/acref-9780199546091.

[9] Shaker, N., Togelius, J. and Nelson, M. J. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* 1st ed. Springer, 2016. ISBN 978-3319427140.

[10] Steffano, V. *The Many Different Types of Video Games & Their Subgenres* [online]. iD Tech, 2018 [cit. 2022-5-1]. Available at: https://www.idtech.com/blog/different-types-of-video-game-genres.

[11] Technologies, U. *Unity User Manual 2021.3 (LTS).* 2022. Available at: https://docs.unity3d.com/560/Documentation/Manual.

[12] Thorn, A. *Game engine design and implementation.* 1st ed. Sudbury, Mass. : Jones & Bartlett Learning, 2011. ISBN 978-1-4496-5648-5.

[13] Watkins, R. *Procedural Content Generation for Unity Game Development.* 1st ed. Packt Publishing, 2016. ISBN 1785287478.

[14] Whittaker, R. H. *Communities and ecosystems.* 2nd ed. Macmillan, 1975. ISBN 0-02-427390-2.