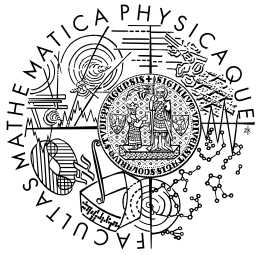


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

**SOFTWARE PROJECT**

Jiří Filek, Martina Fusková, Shivam Sharma

# Dungeon of Chaos

Department of Software and Computer Science Education

Supervisor of the Software Project: Pavel Ježek

Study programme: Visual Computing and Game Development

Study branch: Computer game development

Prague 2023

# Contents

# Introduction

Dungeon of Chaos is a 2D dungeon crawler developed in Unity using C#. The main focus of the game is on stamina-based combat that requires skill from the player. We achieve that by telegraphing the attacks of the enemies and giving the player just enough time to react and make a decision. Thus the pacing is slightly slower than in other games of this genre, such as Diablo. In order to keep the action and increase difficulty, all hits have significant consequences (they deal a lot of damage).

Due to the difficulty, it is possible that the player will die quite often. However, there are going to be checkpoints scattered around the dungeon, which set a new respawn point. This mechanic should therefore help with avoiding repetitiveness and lowering frustration. Moreover, resting at the checkpoint also respawns all enemies, giving the player an option to gain more experience and get stronger before proceeding.

Another important part of the game is character progression through statistics, which is quite common for this genre. The key part of the character progression is a sophisticated skill system with both active and passive skills. The vast majority of the skills is combat oriented, ranging from defensive skills, buffs, and special attacks to ranged attacks and AoE attacks.

Procedurally generated dungeons are the last important part of the game. We are generating them at design time to ensure the high quality of the levels from a layout perspective. It also gives us control over the progression since we are placing enemies, checkpoints, and loot, manually.

## Document Structure

This document is targeted at programmers. The first chapter provides a brief overview of the game and its main mechanics. The second chapter is a high-level overview of the code architecture of the project. The remaining chapters describe core concepts in more detail.

We recommend going through the design documentation first, as it describes design concepts, and the ideas behind them, and the content and its creation in the Unity Editor. It is important to correctly understand why some features are implemented in a particular way. Each feature primarily needs to satisfy the game design. We provide the Game Design Document and the list of Contributions as attachments.

# 1. Project Overview

This chapter provides a brief overview of the project. For more details and a better understanding of the game, read through the design documentation.

The project uses Unity version 2020.3.19f1 LTS. The whole project, including source code, is available at GitHub page. The *Assets* folder is a root-level folder that holds all content files, such as sounds, scripts, or images.

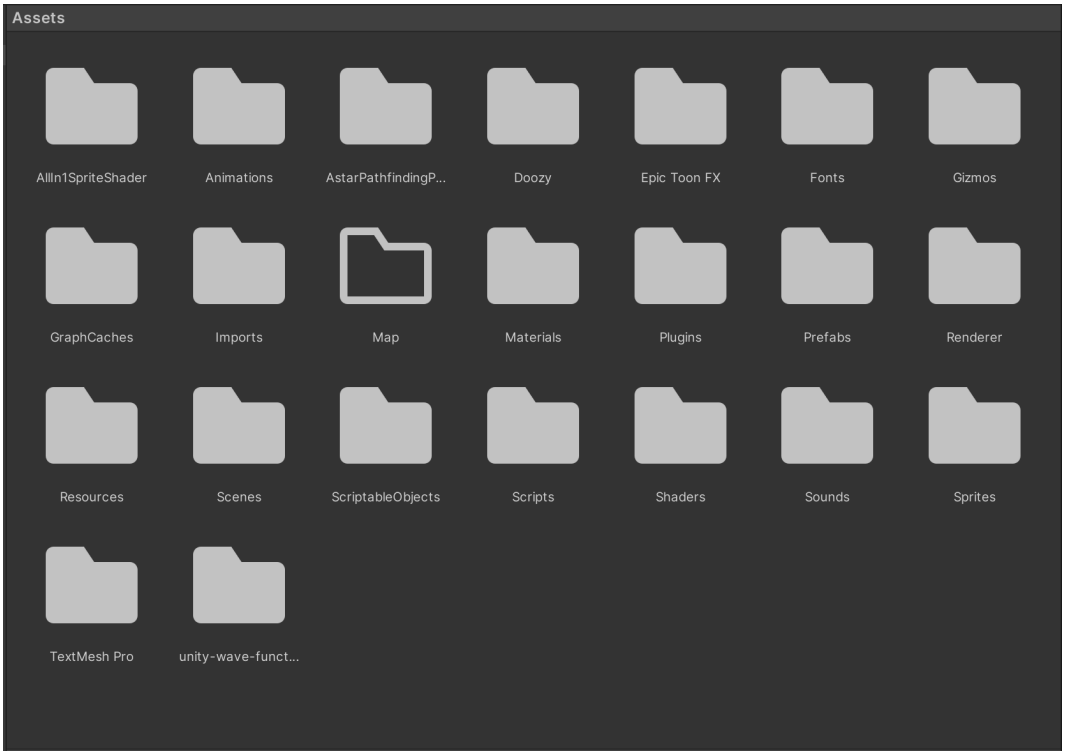


Figure 1.1: The folder structure of the *Assets* folder

Folders in figure 1.1 are divided based on the type of content — all scripts are placed in a single folder. Those folders contain assets appropriate for their names, such as *Scenes*, *Scripts*, *Sprites*, or *ScriptableObjects*. Alternatively, some folders contain external resources — those are $AstarPathfinding$, *unity-wave-function-collapse* (used for the dungeon generation), $AllIn1SpriteShader$ (used for VFX), $DoozyUI$ (used for tweening the UI), $EpicToonFX$ (used for creating VFX such as fireballs, blasts etc.) and $TextMeshPro$.

## 1.1 Scenes

The project has 3 types of scenes: *Gameplay*, *UI-only*, and *Dungeon Generation*.

Each *Gameplay* scene contains one level of the game. The player spends the majority of time in them. *Gameplay* scenes are by far the most complex ones.

We will explain many of their elements in the following chapters.

The *UI-only* screens serve for loading a *Gameplay* scene. Those are the *Main Menu* and the *Game Won*. Both of them are very simple. They contain only UI elements and a few other important pieces. Those are the soundtrack and the sound system itself. The *Main Menu* also contains the save system to read the saved data and communicate them to the player.

The *Dungeon Generation* scene is very special. It is not part of the build. It is used only for generating new dungeons at design time.

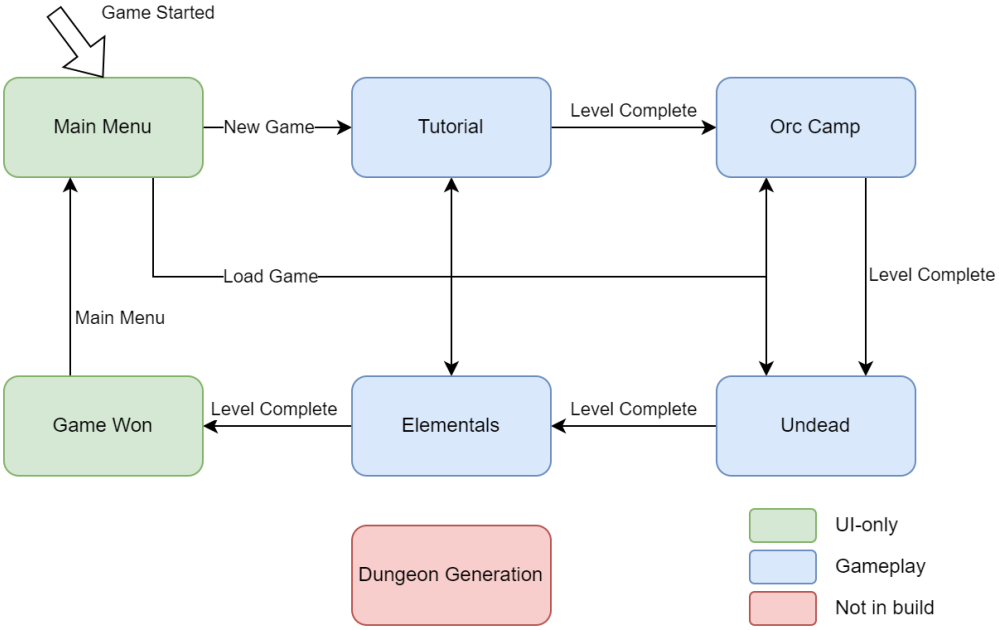


Figure 1.2: Scenes and their transitions

The diagram of all scenes is shown in figure 1.2. The *Main Menu* scene is loaded when the game starts. All *Gameplay* scenes can be loaded from the *Main Menu*.

The first *Gameplay* scene is *Tutorial*, which introduces core mechanics to the player. The remaining three *Gameplay* scenes are *OrcCamp*, *Undead*, and *Elementals*, respectively. Each subsequent level is more challenging than the previous one.

We could transition to any *Gameplay* scene from the *Main Menu*. For example, if the player saves the game in the *Undead* scene, this scene will be later loaded. However, we cannot progress backward in the diagram unless a new save is created. It is common for the player to stay in the same *Gameplay* scene. When they die, the same scene is loaded again.

## 1.2 Gameplay

This section is vital for understanding the game. We will show how the gameplay works and explain the game's core loop. Moreover, we will describe the main interactions of the core loop in more detail. Therefore, we will also present an overview of the implemented features.

Figure 1.3: Gameplay screen; the top image is raw, the bottom is annotated

Figure 1.3 shows how the game looks during the gameplay. To get a better overview, we also included an annotated version. The most important part is in the middle in red color. It shows the player, enemies orc preparing his attack, and a yellow attack indicator. If the player stayed in its area, we would be damaged. The blue color shows the UI. Health, mana, and stamina bars are in the top left corner. At the bottom of the screen, we can see the experience bar

and skills status. The player has only one skill right now and is not on cooldown. The green color represents the dungeon and its objects. Those objects block the movement of the player and cast shadows. Finally, in the yellow color, there are other dungeon objects. In the middle is a checkpoint, the most important dungeon object that allows us to save the game and level up the character. On the wall, there is a torch. It gets lit when the player is nearby.

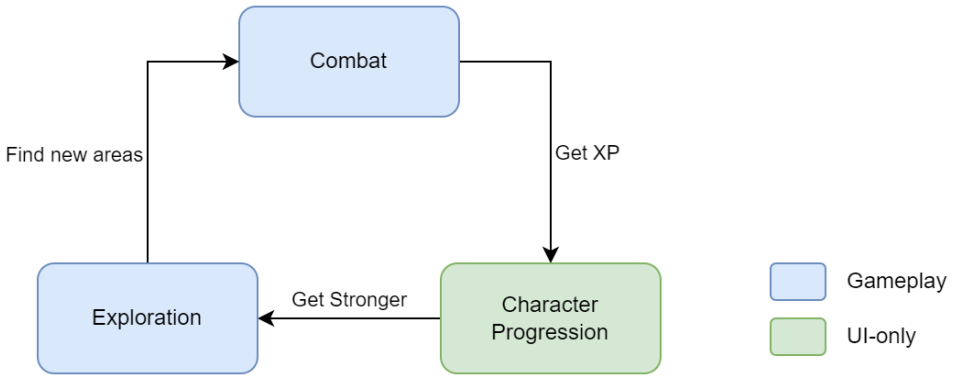


Figure 1.4: Core Loop

Now that we know what the game looks like, we can explain the core loop and the main interactions. The core loop can be summarized by figure 1.4. The goal of the game is to finish all dungeons. In order to fulfill the goal, the player will explore new areas, fight enemies, and gradually strengthen the character. These three parts of the gameplay are repeated throughout the whole game, and one part leads to another, as visualized above.

Figure 1.5 shows the game's flow in a single dungeon in more detail. The player keeps fighting, exploring, and possibly dying. In that case, they will lose progress. Finally, they reach a new *Checkpoint*. Then their progress is saved, and if they have gained enough XP by killing enemies, they can level up the character and learn new skills to strengthen them. Afterward, they can move to a different part of the dungeon to find the boss. At the end of each dungeon, there is a unique boss. When the player manages to beat the boss, the dungeon is finished and they are moved to the next one.

### 1.2.1 Exploration

Figure 1.6 shows the orc dungeon layout. The player is not able to see the whole dungeon. The green line denotes the shortest path to the boss in the bottom left corner. The rest of the dungeon is purely for exploration and level creeping. However, the player does not know where the boss is located. Therefore, they

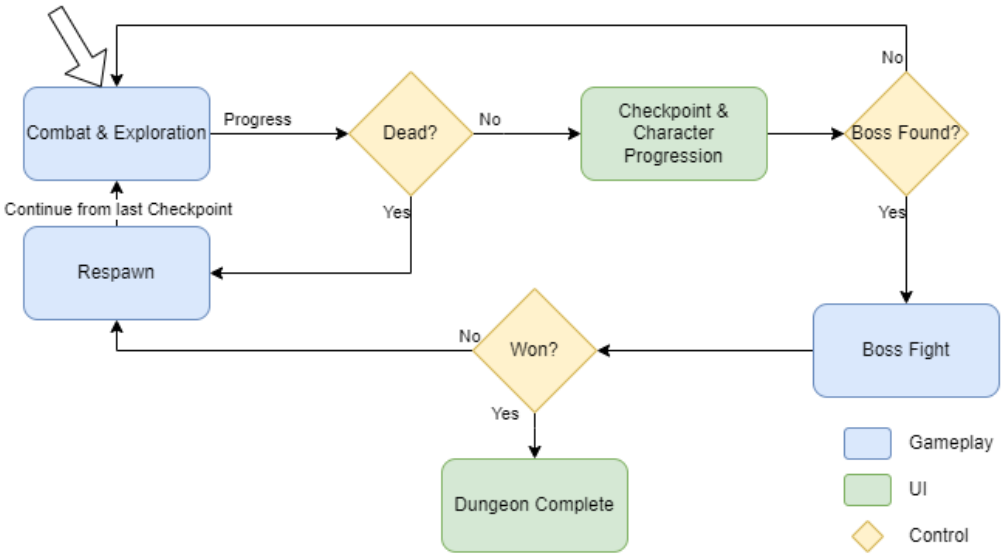


Figure 1.5: Dungeon's Flow

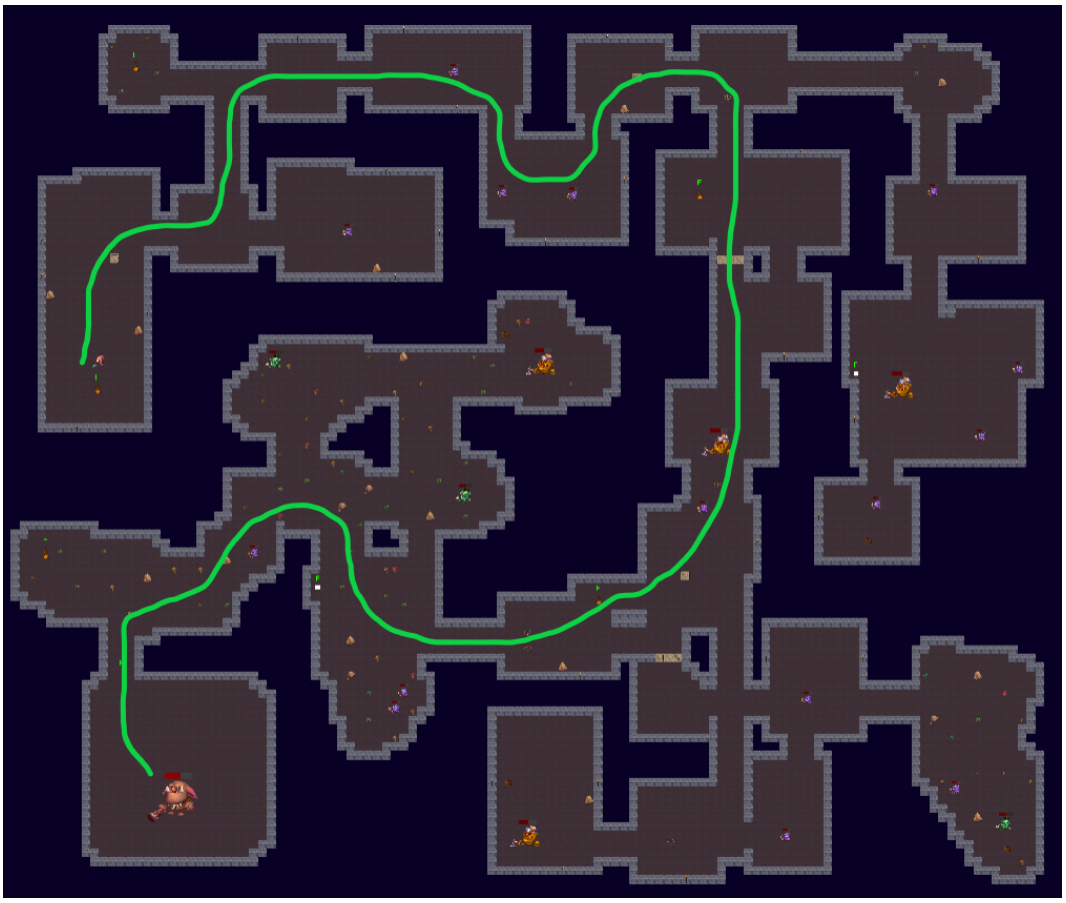

Figure 1.6: Dungeon's Layout

must explore the level to progress.

The dungeon is full of objects the player can interact with. One of the most important objects is the checkpoint shown in figure 1.7. Visiting a checkpoint

saves the player's progress. It also allows them to level up their character. Moreover, it respawns all enemies, giving the player the option to get stronger by defeating them again.

Figure 1.7: Checkpoint

Getting stronger is essential to the game, as the dungeon has many dangers. The player can encounter enemies that are too powerful for them at the moment. There are also traps and lava lakes that deal damage when stepped into.

Figure 1.8: Traps — off and on

However, there are also several mechanics to reward the player for exploration. The first bunch of mechanics is related to the character progression. Aside from gaining experience points by defeating enemies, the player can also obtain them from chests scattered throughout the dungeon. Moreover, they can also collect a book that allows them to redistribute points invested into skills.

Figure 1.9: Dungeon objects — Chest, Map Fragment, and Reset Book

The second mechanic is a minimap that simplifies the navigation in the dungeon. An example of the minimap is shown in figure 1.10. It is not the classical minimap. The player sees only their relative position inside the dungeon and the very close neighborhood. All visited checkpoints are also visible. Moreover, they

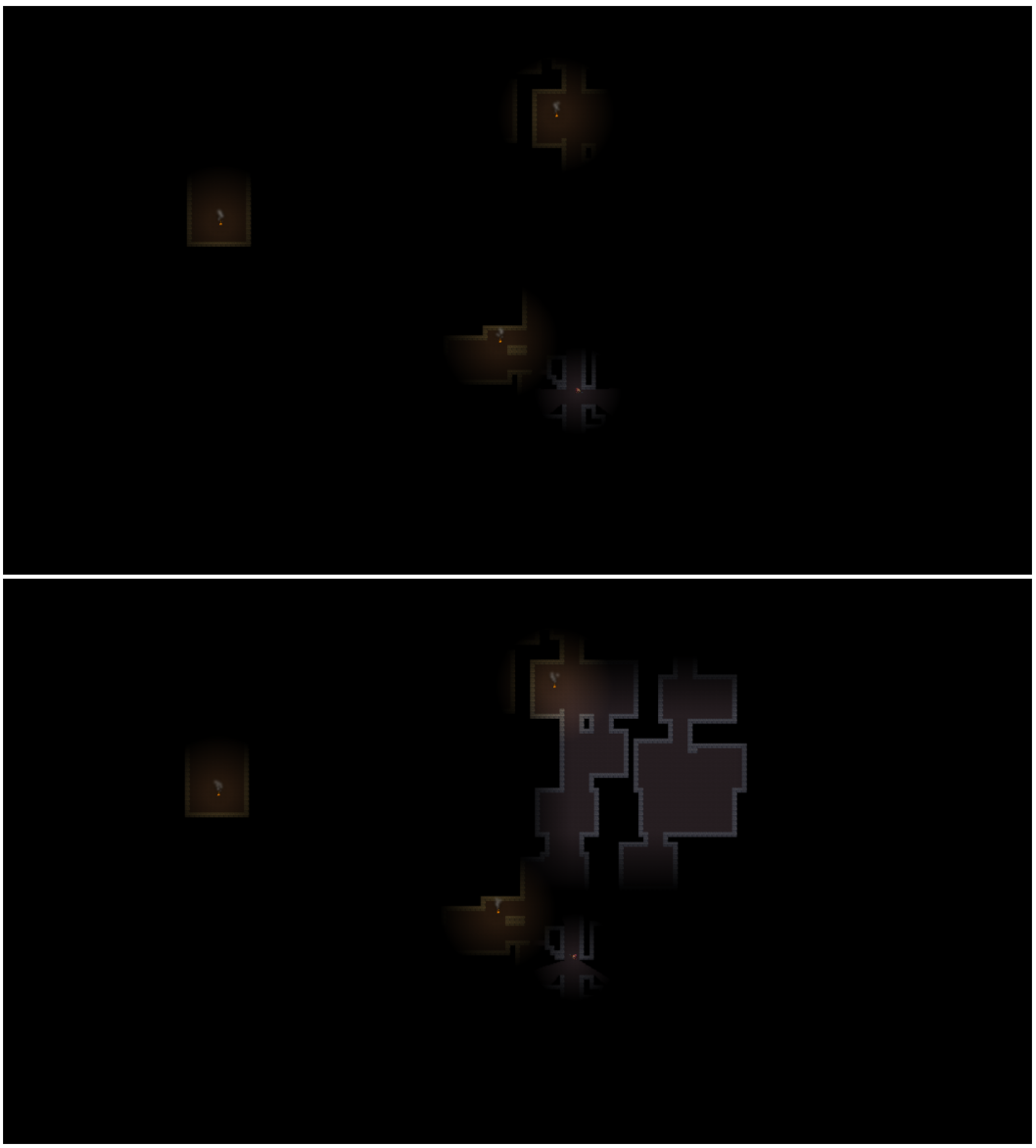

Figure 1.10: Minimap — the top map has only Checkpoints, the bottom map shows a part revealed by the Map Fragment

can collect map fragments that reveal a large part of the minimap. This helps the player to orient better and find the boss faster.

### 1.2.2 Combat

The player needs to beat enemies to progress through the dungeon. The combat is supposed to be skill-based. Therefore, all enemies have telegraphed attacks so the player can avoid them. Both the player and the enemies can attack in all directions. The player's weapon follows their cursor and will attack in the current direction. To make the combat more interesting, the player has more types of attacks, and they can dash. These actions cost stamina, an important currency

during combat, as low stamina can leave the player defenseless. The player can also use spells during combat, which cost mana.

Figure 1.11: Combat encounter

Figure 1.11 shows an example of an encounter in the dungeon. We can see there is a yellow cone under the player. It serves as an indicator of an attack by the big yellow orc. The player can still dodge the attack without being hit. This approach was selected since indicating attacks by animations would be very costly art-wise. We can also see a health bar over each enemy.

There are many unique enemies in the game. They differ in strength, attack types as well as visuals or theme. The exact content is described in the GDD.

### 1.2.3 Character Progression

The last part of the core loop is the character progression through statistics, as is typical for the dungeon crawler genre. In our game, the player can progress in two systems — stats (such as damage, hp, mana...) and skills (such as a new type of attack, spell, or a passive skill).

It is quite typical to include also some kind of a gear system, however, due to the smaller scope of our game, we have decided not to. Therefore, the only loot the player can collect are consumables that immediately replenish hp, mana, or stamina, experience points, or items that can be used for resetting points invested into skills.

The stats system is common for both the player's character and enemies. Though, unlike the player's character, enemies cannot level up. The stats the player can level are focused only on the combat part of the game, and they affect how much damage attacks or skills deal, how powerful spells are, or how much health, mana, or stamina the character has.

The skill system is used solely by the player's character, however, some of the skills inside of the system can be used by the enemies as well. The skills are also focused mainly on combat, as they give the player more actions they can do, or provide a passive bonus to the stats.

# 2. Code Architecture

This chapter discusses the overview of essential systems. It means a high-level point of view.

Figure 2.1 shows the most crucial systems and their dependencies. Each system is implemented in several source files. We will deal more deeply with each of them in the dedicated chapters.

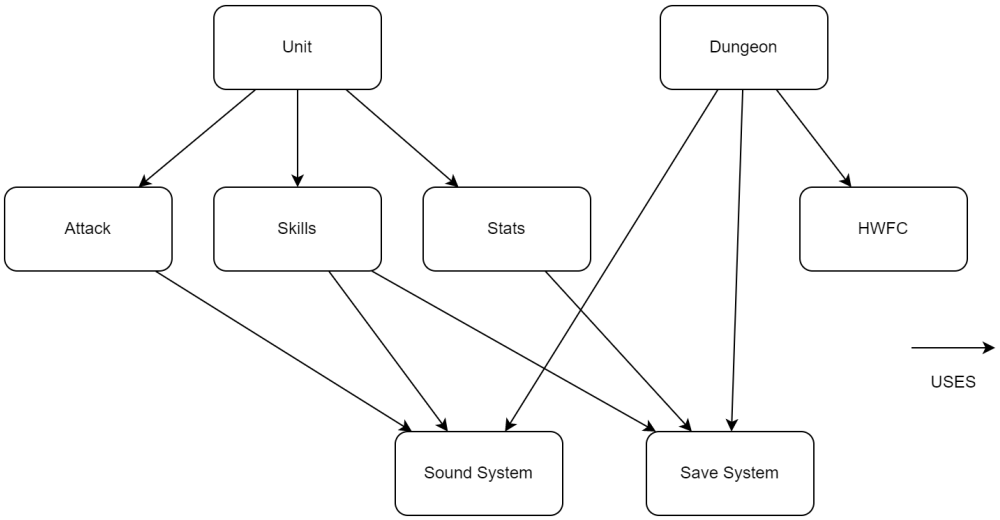


Figure 2.1: Systems Overview

We will start from the top systems in figure 2.1 and go down. Therefore, we will describe essential systems for gameplay first. There are Units – the player's character and enemies running around the dungeon and attacking each other. Those are built on top of other systems. Under them, we have Attacks, Skills, and Stats, which define the gameplay and the RPG interactions – for example, damage calculations. At the lowest level, we have stable systems – the Sound and Save systems. Independently of the Units, we have a Dungeon. The Dungeon is generated using the Hierarchical Wave Function Collapse algorithm (HWFC). It also provides many gameplay features, such as checkpoints, traps, or chests with loot. The following chapters focus primarily on a technical point of view.

All of those systems influence the gameplay. Therefore, they are all part of the main gameplay scenes. The source code is placed in the *Scripts* folder. We use $Clang-format$ with the $Microsoft$ preset to make the code formatting nice and consistent.

# 3. Units

Units are a crucial part of the game. They provide a majority of gameplay. We have two types of units: the player's character and various enemies.

Units are relatively complex. The `Unit` is a base class for `Character` and `Enemy`. We heavily utilize composition to split responsibilities among different systems for multiple parts of each unit.

The unit has the following important members

- `Stats`
- `IMovement` — how should the unit move
- `IEffects` — special effect when taking damage
- `IBars` — health, mana and stamina bars

The `Stats` define all RPG interactions. We will discuss them more in the following chapter. The rest of them are base classes that define given behavior. Implementation-wise, those are behaving as interfaces. However, they are abstract classes deriving from `ScriptableObject`. This way allows us to assign them directly using the inspector. It is not possible for `interface`.

All of those abstract classes have an `Init()` to store the data they will need. We are also creating a new instance of those classes at `Start()`. Therefore, those don't have to be stateless. Otherwise, we could affect the behavior of other units from the nature of scriptable objects.

A simpler alternative would be to derive those classes from `MonoBehaviour` and attach them to our game objects. However, we can easily see which components must be set with our approach. Also, it is all in one place, which is convenient.

Besides that, the unit contains members for sound effects.

Important function:

- `void TakeDamage(float value)`
- `void Die()`

## 3.1 Movement

The movement of units is solved using forces that are applied to `RigidBody2D`. This way, we can avoid glitching units into walls. It would happen when we move them directly by code. The reason is, that the rendering loop is more frequent than the physics loop. Therefore, the physics of collisions might not be yet resolved.

The movement is defined in the `IMovement` class. It contains an important function `virtual void Move()`, which is called in `FixedUpdate()`. Our approach is very flexible, and we could easily add different movements for some special units.

We read and use the player's input for the character's movement. It is a bit more complex for enemies since they use A-star pathfinding to follow the player and avoid walls.

## 3.2 Bars

Bars are essential to communicate the game state to the player. Each unit has a health bar. The character also has a mana bar, a stamina bar, an experience bar, and possibly an armor bar.


Figure 3.1: Status bars

Bars work differently for the player, enemies, and bosses. Therefore, the `IBars` class provides an interface. It has three implementations, one for each type of unit.

The bars of the player are placed in the top left corner of the screen. The experience bar is the yellow bar placed at the bottom on top of the screen. The same applies to the boss's health bar. The health bar of each enemy is on top of it with a constant offset. We can see it in figure 3.1. An important feature of enemy bars is that it uses a lit material. Therefore, like enemies, they are only visible when lit and do not glow in the shadows. When some value changes, it gets propagated to the bars.

## 3.3 Weapon

Each unit has a weapon. The weapon rotates towards the mouse of the player. It works the same for enemies, but they try to rotate it toward the character.

It is represented by a `Weapon` class. The `RotateWeapon(Vector2 target)` handles the 2D rotations. We need the weapon to be properly rotated, so this function is called in `Update()` unless the unit is attacking. We disable the rotation of the weapon during the attack as a design decision. This script is placed in the unit's center and rotates the asset, which is a different object.

Figure 3.2: Weapon rotations

The sprites of units will flip to face the same direction as the weapon. It allows us to have all rotations of the weapon, and it still feels natural. An example of three weapon rotations is shown in figure 3.2.

The weapon is used mainly for attacks. We will deal with them in chapter 6. We want the weapon to hit each target only once during the attack. Therefore, the weapon holds a list of units that were already hit. The hit resolution is made in `HitChecker`. It is a trivial script attached to the asset of the weapon. It detects the collision and is used as a callback to the `Weapon`.

The weapon belongs to the arm of the unit. However, we need to set a default angle for a neutral position. It is required for attacks to know this offset.

Otherwise, the weapon would move unnaturally, or all weapons would need to have the same default angle.

## 3.4 Character

The `Character` derives from `Unit`. It is crucial since it is the main class used by the player. It is a singleton since the player is important for many systems.

The character has its `SkillSystem`. We will discuss it more in chapter 5. It has many important functions for attacking, rotating the weapon, or using skills. However, most of those functions are private. Those functions are pretty simple since most of the work is done in some system.

The public API is very simple. There is only one function that allows the blocking of the input. It is required for UI screens to block the interaction of the character. Besides that, we can get the stats of the character.

The character takes damage when an enemy touches him. We can it a bite attack. This mechanic is used to make melee enemies more dangerous. It is solved using `OnCollisionEnter2D()` and cooldown since the last bite.

## 3.5 Enemy

`Enemy` is another class that is extended from `Unit`. This class is the key for the enemy units as it handles all the key functions of the Enemies in the game.

The class is responsible for the detection of the target, rotating between the various enemy states such as Idle, Attack, and Chase, and choosing the best attack to perform against the player. Almost all of these functions are private.

The enemy takes damage when hit by the player's weapon, or other skill attacks that the player may have. There are four different types of enemies based on their power levels and attacks. While the enemies are explained further in the next section, the attacks would be discussed in detail in chapter 6.

### 3.5.1 Enemy Types

Based on the various parameters such as HP, attacks, speed, and, placement, enemies can be broadly classified into four different types. These types will be discussed in detail below.

#### Basic

As the name suggests, this is the most basic or the weakest type of enemy. It usually has very a single attack with very low damage. Their biggest strength is

that they are in abundance throughout the map and can gather up against the enemies.

#### Elite

Elites are a level up compared to the basic enemies. They are stronger, have more health, and inflict higher damage.

#### Mini Boss

Mini-Bosses are powerful enemies with significantly increased health. They have multiple attacks in their arsenal and can be very quick. They usually have their own dungeon room and come later in the game.

#### Boss

Bosses are the most powerful enemies. They are placed at the very end of the map. They have the most powerful attacks including special attacks that can summon other enemies, as well as heal them. Bosses also have the most amount of health and can pose quite a challenge to the players.

### 3.5.2 Enemy AI

Enemies in the game are capable of detecting if the player is in vicinity, chasing it and if available choosing the best attack to use on the player. The AI switches between three states - Idle, Chase, and Attack.

The default state of the enemy is the idle state. During this state, the enemy doesn't move but fires raycasts to detect if the player is in the chase range. If it detects the player, the state changes to chase. In this state, the enemy constructs a path towards the player and move towards it. While chasing the player, the enemy also check if the player is within the attack range of any of the avaialble attacks. If an attack is available to be performed, the enemy state changes to attack and it performs the attack on the player. The enemy AI is made up of two parts - Movement, and Combat which are discussed in detail up next.

#### Enemy Movement

For the enemy movement, a third party unity plug-in called A* Pathfinding Project by Aron Granberg has been used. It is used to bake the navigation grid as well as calculating the path. Although, the two major methods that are used from the public API of this plug-in include `UpdatePath(Transform targetT)` and `UpdateMovement(Vector3 separationForce)`. The first method calculates the

path from the target and updates the waypoints everytime it is called. The latter method is used to drive the enemy unit using `rg.AddForce(Vector3 Force)`. The parameter `Vector3 separationForce` is used to steer the enemy around the obstacles if any.

For the proper functioning of the movement AI, it is important to add the component `AIAgent` to the same `GameObject` that includes the Enemy `Rigidbody`.

Figure 3.3: Skill System Overview

As depicted in the Figure 3.3, the baked grid includes blue and red nodes. The blue ones are accessible and are used in constructing the path waypoints. The red ones form the perimeter of the walls as well as the obstacle and are ignored while calculating the path. The green curve depicts the path from the enemy to the player during that particular frame.

### Enemy Combat

The more powerful enemies have more than one attack in their arsenal. The component `AttackManager` is responsible for selecting the best available attack for the enemy to perform. `AttackManager` has access to all the `IAttacks` that the enemy may have. At the start of the life cycle of an enemy unit, `AttackManager` calculates the priority of all the `IAttacks` based on their damage, range, and cost. For more complexity, each one of these parameters can be assigned a weight in the priority calculation. The attacks are then sorted based on the calculations. The API of `AttackManager` includes the method `IAttack GetBestAvailableAttack()`. The method iterates through the sorted list of attacks and returns the first attack that is available at that moment.

# 4. Stats

The `Stats` influence all interactions between units in the game, and they are an important part of the character progression. We distinguish between two types of stats – primary stats and secondary stats.

Primary stats can be leveled up by the player and are used for calculating the secondary stats. Secondary stats are then directly used in the game for calculating damage or results of other interactions.

There are the following stats in the game:

- Physical Damage – secondary stat used for calculating damage of physical attacks and skills
- Spell Power – secondary stat used for calculating the value of spells
- Health – secondary stat determining the maximum health
- Mana – secondary stat determining the maximum mana
- Stamina – secondary stat determining the maximum stamina
- Movement Speed – non-levelable stat determining the movement speed
- Stamina Regeneration – the amount of stamina regenerated per second, secondary stat
- Armor – the amount of armor the unit has after respawning, secondary stat
- XP Gain Modifier – modifier used for calculating the amount of XP gained
- Cooldown Modifier – modifier used for calculating the cooldown of skills
- Stamina Cost Modifier – modifier used for calculating the stamina cost of skills and attacks
- Mana Cost Modifier – modifier used for calculating the mana cost of skills

Some of the stats above can be leveled up directly by the player, some can be improved only through skills, and some cannot be improved at all. The design perspective of stats is described in more detail in the GDD.

The `Stats` is a class derived from the `ScriptableObject`. This way new instances of `Stats` can be easily created for new units, and then assigned in the inspector to the corresponding unit.

Health, mana, and stamina are of type `RegenerableStat`. This means, that they have a current value and a maximal value. For this purpose, we have created

the `RegenerableStat` class. The class holds the two values and it also handles all operations on the stat – such as regenerating, consuming, or resetting.

Aside from the stat variables mentioned above, the class has the following important members:

- `PrimaryStats` – stats the player can level up, the secondary stats are calculated from them
- `IBars` – a reference to health, mana, and stamina bars
- `Levelling` – leveling data of the unit, such as current level or amount of XP

All the members are `private` and the `Stats` class has `public` methods for getting and setting the data as is needed by other systems – such as skill system, character sheet UI, or units.

## 4.1 Primary Stats

Primary stats can be leveled up by the player and they are used for calculating the final secondary stats. One primary stat can determine more secondary stats.

We have the following primary stats:

- Strength – used for calculating the physical damage
- Intelligence – used for calculating the spell power
- Constitution – used for calculating health and armor
- Endurance – used for calculating stamina and stamina regeneration
- Wisdom – used for calculating mana

The constants used in formulas for calculating secondary stats are defined in the `Primary To Secondary` `ScriptableObject`. Currently, we use only one object for all units, but this implementation allows us to simply have one object for character and one object for enemies, or even have a unique object for each unit.

The formulas are then defined in the `PrimaryStats` class itself for each secondary stat. These functions are `public` so they can be called from the `Stats` class.

For enemies, their primary stats are modified before they are used for calculating the secondary stats, based on their power type (basic, elite, boss) and

their level. This makes it easier for designers to balance the values of enemies. Moreover, it allows us, to define values for some enemy, and then easily create several instances with different levels.

The constants used for calculating the modifications are defined in the `StatsModifiers` `ScriptableObject`. The formulas are again defined in the `PrimaryStats` class itself.

## 4.2 Leveling

The `Levelling` class is responsible for holding all data related to leveling and is also responsible for handling leveling up. The class is used mainly by the player's character as enemies don't level up. The only thing enemies use from this class is the `level`.

We can set several values in the inspector that determine how the character progression is going to work:

- `maxLevel` – determines the maximal level the player's character can reach
- `baseXP` – determines the amount of XP needed for the first level
- `baseMultiplier` – determines the multiplier that is used for the growth of XP needed for the next levels

Otherwise, the class also contains variables representing the current state of character progression:

- `level` – the current level of the character, must be above zero and below max level
- `statPoints` – can be invested into leveling up the primary stats
- `skillPoints` – can be used for unlocking or leveling skills
- `currentXP` – amount of XP the character has, it can be larger than `nextLevelXP` as the player needs to level up manually
- `nextLevelXP` – amount of XP needed for the next level, it grows exponentially with the level

Most of the functions are getters and setters of the variables mentioned above. It also handles the logic behind leveling up and showing correct leveling data inside of the UI.

# 5. Skills

Skills are an extremely important part of the game, as they provide the player's character with more actions they can do during combat. The skill system is also a huge part of the character progression, one of the core parts of the whole game.

The design of the skills, a list of implemented skills, and a guideline for creating new skills in the Unity Editor can be found in the GDD. This chapter provides only an overview of the classes and their relations from the coding perspective.

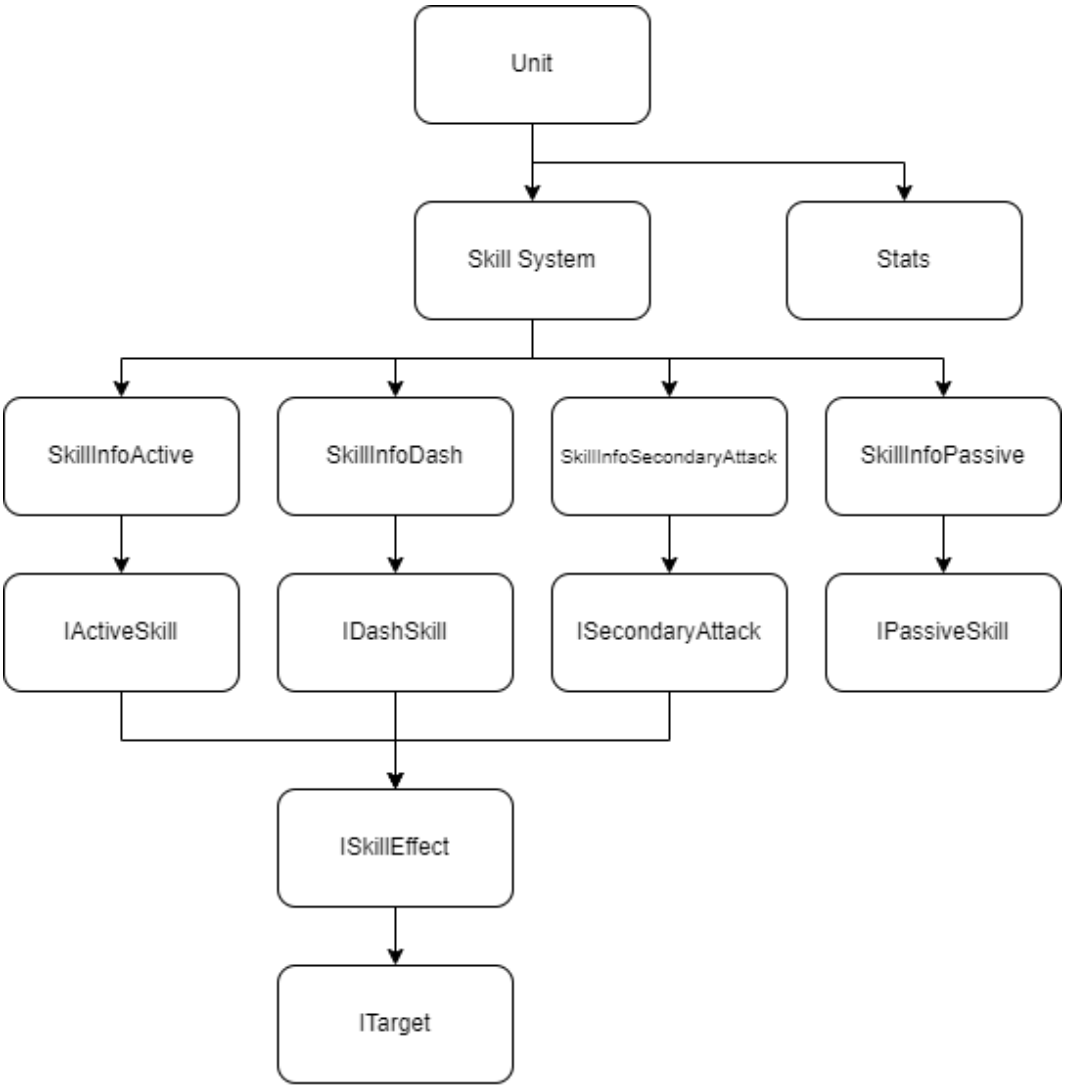


Figure 5.1: Skill System Overview

Figure 5.1 shows the architecture of the skill system and its dependencies on other classes. Starting at the top, each `SkillSystem` class has its owner `Unit` which can then use the skills with values depending on the `Stats`. The `SkillSystem` class then holds all skills and handles operations on them. The class is described in more detail in 5.1.

The skills themselves are then implemented in three layers – `SkillInfo`, `ISkill`, and `ISkillEffect`. The `SkillInfo` layer holds all levels of skill and data and functionality required for leveling. The `ISkill` layer represents one level of skill, holding the behavior of the skill and its description. The `ISkillEffect` layer implements a small part of the behavior of the skills which allows simple reusability. The composition of skills and individual layers are described in more detail in the following sections of this chapter.

## 5.1 Skill System

The `SkillSystem` class holds all skills that are present in the game. It is also responsible for handling all operations on skills, such as unlocking, resetting, activating, or using.

It contains a list of all skills (a list of `SkillInfo<T>`) per each skill type. There are four skill types in the game – active, passive, dash, and secondary attacks. The individual types will be described in more detail later. The rest of the class, as well as other classes interacting with `SkillSystem`, work with an index pointing to one of those lists.

### 5.1.1 Active Skills

Active skills need to be activated after they are unlocked in order to be then used in the game. Skills are activated by dropping them into one of the available skill slots in the UI.

The `SkillSystem` holds a list of activated skills as indices into the complete list of active skills. And the following public API is implemented for operations with active skills:

- `void UseSkill(int index)` – uses a skill at the given index in the list of activated skills
- `bool CanUpgradeActive(int index)` – returns `true` if the player can upgrade the skill at the given index in the list of all active skills
- `void UpgradeActive(int index)` – upgrades the skills at the given index in the list of all active skills
- `void Activate(int index, int slot)` – activates the skill at the given index and places it at the correct index in the list of activated skills based on the slot

- `bool IsUnlockedActive(int index)` – returns `true` if the skill at the given index in the list of all active skills is unlocked
- `bool IsActivated(int index)` – returns `true` if the skill at the given index in the list of all active skills is activated

### 5.1.2 Dash Skills

Dash skills are in their behavior very similar to active skills and the public API is also very similar to active skills. However, they have only one slot available, therefore instead of a list of activated dash skills, the `Skill System` holds only one index. Moreover, the character has the basic dash unlocked from the beginning of the game.

The `SkillSystem` implements the following public API for dash skills in addition to the functions mentioned for active skills.

- `void Dash(Vector2 dir)` – the character dashes in the given direction
- `bool IsDashing()` – the character cannot dash if it is already dashing
- `bool IsAttackDashing()` – a separate function is needed for attacking dash which deals damage to enemies on collision because in that case, we don't want to deal damage to the character
- `void DashCollision(Collision2D col)` – the character stops dashing on collision

### 5.1.3 Secondary Attacks

Secondary attacks are very similar to dash skills, as they also have only one slot available and are otherwise the same as active skills. The only difference between secondary attacks and dash skills is, that there is no secondary attack unlocked at the beginning.

When it comes to the public API, there are no special functions for secondary attacks, except for `bool IsAttacking()` which is used so the character cannot use both primary and secondary attacks at once.

### 5.1.4 Unlocking

The `SkillSystem` is also responsible for handling the unlocking of the skills and checking whether a given skill can be unlocked. There are various unlocking requirements, such as the number of skill points, the level of the character, values of primary stats, or other skills.

The functionality is implemented using generic methods so it can take as a parameter skill of any type. It has a constraint to the `ISkill` class which is a base class for all skills.

It has a public function `bool CanUnlock<T>(SkillInfo<T> skillInfo)` which then checks whether all unlocking requirements are fulfilled. Each requirement has its own function which is private.

### 5.1.5 Resetting

Skills can be reset by using a reset book. Resetting gives the player back all invested skill points and locks all skills and sets their levels to zero.

Code-wise it is very simple, as it simply goes through each list of skills and locks each skill. Moreover, it also resets the list of activated skills to its default values.

## 5.2 Skill Composition

In order to support simple extendability, skills of various types, and a quite wide range of behaviors, we have decided to split the skills into several layers. These layers and their functions slightly differ based on the type of skill, but some are common to all of them.

The first layer is the `SkillInfo` layer which holds all levels of the skill, unlocking requirements for each level. It also holds basic information about the skill and all functions that are needed by the `SkillSystem`. It will be described in more detail in 5.3.

The second layer is the `Skill` itself. This layer is responsible for holding data about a given level of the skills, such as name, description, or icon. On this layer is also implemented some part of the behavior of the level of the skill, or possibly even the whole behavior, that depends on the type of the skill.

The last layer, `SkillEffect`, is used mostly only by active skills, dash skills, and secondary attacks. The second layer usually has a list of `SkillEffect` that should be applied when the skill is used. The `SkillEffect` then implements some simple behavior, thus most skills have usually several effects.

## 5.3 Skill Info

The `SkillInfo<T>` is a generic class where `T` is a constraint to the `ISkill` type. It derives from `ScriptableObject` so we can easily create new skills and

assign them to the inspector. Moreover, it serves as a base class for `SkillInfo` classes of each skill type.

The class is responsible for holding all levels of skill, their unlocking requirements, descriptions, and other information. The `SkillSystem` stores the `SkillInfo<T>` as it contains all information and functionality needed for unlocking new skills or showing them in the UI.

For unlocking new skills and working with them in the `SkillSystem` it implements the following functions:

- `UnlockingRequirements GetUnlockingRequirements()` — gets the unlocking requirements for the next level of the skill. If they don't exist, it creates the default ones.
- `bool IsUnlocked()`
- `void Unlock()`
- `bool CanUpgrade()` – checks whether max level was not reached
- `T GetCurrentSkill()` – returns currently unlocked skill
- `List<T> GetSkills()` — returns a list of skills (one skill per level)

All information about the state of the skill, such as whether the skill is unlocked or on which level the skill is, about the functionality of the skill, and about unlocking requirements, need to be presented in the UI for the player. The `SkillInfo` implements the following functions to facilitate that:

- `SkillData GetSkillData()` – returns data such as name, icon, and basic description of the skill
- `SkillDescription GetCurrentDescription()` – returns description and stamina/mana cost
- `SkillDescription GetNextDescription()` – returns a description of the next level of the skill, including the unlocking requirements

Skill points invested into the skills can be reset by using the reset book. In order to do this, the `SkillSystem` needs to know the number of skill points that were invested and it needs to have a way to reset the skill into the initial setting:

- `void ResetLevel()` – sets the level to 0 and the skill to a locked state
- `int GetInvestedSkillPoints()` – returns the amount of invested skill points based on the unlocking requirements of the skill

Each type of skill – active, dash, passive, and secondary attack, has its own class extending the base class. These classes are used for creating the `ScriptableObject` of the respective skills.

- `SkillInfoActive` extends the class `SkillInfo<IActiveSkill>`
- `SkillInfoDash` extends the class `SkillInfo<IDashSkill>`
- `SkillInfoSecondaryAttack` extends the class `SkillInfo<ISecondaryAttack>`
- `SkillInfoPassive` extends the class `SkillInfo<IPassiveSkill>`

`SkillInfoActive`, `SkillInfoDash`, and `SkillInfoSecondary` all implement only the `public void Upgrade()` function which increases the level of the skill.

`SkillInfoPassive` class implements in addition also functions for equipping and unequipping the skill:

- `public void Equip()` – puts the correct level of the passive skill into effect
- `public void Unequip()` – cancels the effects of the current level of the passive skill

## 5.4 ISkill

The `ISkill` class is used for storing data and functionality of one level of the skill. It is an abstract class deriving from the `ScriptableObject`. It has a `SkillData` field which contains the name, icon, and description of the skill, and a getter for this field.

It also has two abstract public functions `string GetEffectDescription()` and `string GetCostDescription()` which return the functionality description, and stamina and mana cost description, respectively.

### 5.4.1 IActiveSkill

The `IActiveSkill` extends the `ISkill` base class and as the name suggests, it is used for the active skills. The most important function is the `void Use(owner, targets, targetPositions)` which first checks whether the skill can be used (if the player has enough stamina, mana and the skill is not on cooldown) and then it applies all `ISkillEffects` on the target units or target positions.

The `IActiveSkill` overrides the `string GetEffectDescription()` and `string GetCostDescription()` functions.
The `string GetEffectDescription()` goes recursively through all skill effects to gather their dynamic values and add them to the description defined by the designer using `string.Format()`.

### 5.4.2 IDashSkill

The `IDashSkill` extends the `IActiveSkill` class and is used for creating dash skills. It overrides the `void Use(owner, targets, targetPositions)` so the functionality of the `Dash` class is used.

It serves as a wrapper over the `Dash` class which implements the actual behavior of the dash using the following functions:

- `Dash Init(speed, effects, trailColor, unit)` – initialization of the Dash object so it can be used by the Dash skill
- `virtual void Use()` – a virtual function responsible for calling all behaviors needed, such as the `private IEnumerator DashAnimation(Vector2 dir)`. The Dash animation then visualizes the trail, plays the sound effect, and applies force to the owner unit.
- `virtual void OnCollisionEnter2D(Collision2D col)` – resolves collisions and applies skill effects

The base version of dash is used by dash skills without skill effects or with skill effects that are applied on collision. `PositiveEffectDash` is a subclass of `Dash` which overrides the `void Use(Vector2 dir)` and `void OnCollisionEnter2D(Collision2D col)` functions. In this case, skill effects are applied on the owner unit when the dash is used. This is used for example by the Healing Dash skill.

### 5.4.3 ISecondaryAttack

The `ISecondaryAttack` extends the `IActiveSkill` class and is used for creating secondary attacks. It overrides the `string GetEffectDescription()` function as it only needs the data from the `AttackConfiguration`. Moreover, it overrides the `void Use(owner, targets, targetPositions)` function to use the functionality of the `IAttack` class.

The secondary attacks are using the implementation of regular attacks for their behavior. Each secondary attack has a game object with a component derived from the `IAttack` base class, and a `AttackConfiguration` as fields. The type of the `IAttack` component depends on the type of secondary attack. When the player equips a secondary attack, the component is added to the weapon of the character and is initialized with the configuration using the `void Init(owner)` function.

When the player unequips the secondary attack, the component with the secondary attack on the character's weapon is destroyed by the `void Deactivate()` function.

### 5.4.4 IPassiveSkill

The `IPassiveSkill` class is an abstract class that extends the `ISkill` base class. It contains two abstract functions `void Equip(Stats stats)` and `void Unequip(Stats stats)`. These two functions are then implemented in the subclasses of this class.

The base class also overrides the `string GetCostDescription()` function by returning an empty string since passive skills don't have any cost or cooldown.

#### Increase Stat

The `IncreaseStat` is an abstract class extending the `IPassiveSkill` base class. Passive skills derived from this class are affecting the stats of the character, such as armor, damage, spell power, cooldown modifier, or xp income. This is done using the abstract function `void ChangeStat(Stats stats, float val)` whose implementation depends on the stat that is supposed to be changed.

In `void Equip(Stats stats)` the value of the bonus is calculated based on the amount specified by the designer, and it is scaled with the level of the character. Afterward, the `void ChangeStat(Stats stats, float val)` is called with this value. `void Unequip(Stats stats)` simply calls the same function with the negative value.

#### Regenerable Stat Improvement

The `RegenerableStatImprovement` is an abstract class extending the `IPassiveSkill` base class. Passive skills derived from this class are used for improving mana and stamina, namely increasing their regeneration and decreasing skill costs. Similarly to `IncreaseStat` class, it implements a

`void ChangeStat(Stats stats, float reg, float c)` which changes the regeneration values and cost modifier.

#### SecondBreath

The `SecondBreath` class extends the `IPassiveSkill` base class. This passive skill is different from those that increase some stat immediately after equipping as its effect is triggered later. For this reason, it uses a skill effect `Resurrect` which heals the character after they die for the first time.

It contains a `bool ShouldResurrect()` function which is called every time the character dies. The `SkillSystem` is responsible for checking whether the skill is unlocked and thus should be used.

The effect of the resurrection is handled in the `void Resurrect(Unit unit)` function which plays a VFX and SFX. The VFX then calls the `Resurrect` skill effect that handled the logic.

## 5.5 Skill Effects

Skill effects are used mainly by active skills and they are used for implementing small parts of the behavior of the skills, such as dealing damage, healing, improving stats, or playing VFX and SFX. Skill effects are applied by skills, other skill effects, or visual effects spawned by skill effects.

The `ISkillEffect` is an abstract class that extends the `ScriptableObject` so new skill effects can be created easily. It contains `ITarget target` field for specifying the target or group of targets of the effect. The field is optional as the skill effect can receive its target from the caller.

The class implements a `public void Use(unit, targets, positions)` that is called by the parent skill or parent skill effect. The targeting parameters are optional as they can be specified by the `ITarget target` field.

Furthermore, the class has three virtual functions that are used for applying the effect. Their usage depends on the target:

- `void Apply(Unit unit)` is used when the target is specified by the field of the class
- `void ApplyOnTargets(unit, targets)` is used when the caller passes a list of units
- `void ApplyOnPositions(unit, targetPositions` is used when the caller passes a list of positions

All skill effects then extend the `ISkillEffect` base class and implement at least one of the three `Apply` functions. They override the `string[] GetEffectsValues(Unit owner)` either returning the array of values tied to the effect or recursively going through all skill effects they call to construct the array. Most skill effects with the exception of the ones handling visuals require a `SkillEffectType` for scaling the value with either physical damage or spell power.

### 5.5.1 Visuals

Skill effects can be also used for handling visual and sound effects of the skills. For that purpose, there are three types of skill effects, the `VisualsAndSounds`, `AoEVisual`, and `TemporalVisuals`.

The `VisualAndSounds` class is used mainly for casting spells VFX. It contains a `GameObject vfx` that has the `VisualEffects` component and a particle system with the actual visuals, `SoundSetting sfx`, and a `List<ISkillEffect> effects` that are applied after the VFX finishes playing.

The `AoEVisual` class is used for visualizing skill effects with AoE targeting. It contains a `GameObject vfx` that has the `AoEVisualEffect` component and a particle system with the actual visuals and a `List<ISkillEffect> effects` that are applied to targets inside the AoE. The `AoEVisualEffect` class handles the correct visualization of the AoE in terms of the range of the skill.

The `TemporalVisuals` class is used for visualizing skill effects that apply some temporal effect (such as stat boost or burn) on the target units. It contains a `GameObject vfx` with `TemporalVisualEffect` component and a particle system, and a `List<TemporalEffect> effects`. `TemporalVisualEffect` class can also temporarily change the material of the target unit or its weapon. It applies all the temporal effects from the list that is passed by the `TemporalVisuals` skill effect. The game object is destroyed once all `TemporalEffects` from the list end.

### 5.5.2 Temporal Effects

The `TemporalEffect` is an abstract class extending the `ISkillEffect` base class. It is used for skill effects that temporarily apply other skill effects, such as various boosts or damages over time. There are two types of `TemporalEffects` based on whether the effect is applied only once at the beginning (`OneTimeTemporalEffect`) or repeatedly with some frequency (`RepeatedTemporalEffect`).

Aside from unique visuals handled by the `TemporalVisuals` skill effect and `TemporalVisualEffect` class, each `TemporalEffect` is also associated with an icon that is shown under the bars of the character.

The behavior of the effect is implemented in the abstract `void ApplyEffect()` function. The duration and prospective repeated application of the effect are then handled by the `public abstract bool Update()` function that is called in the `void Update()` function of the `GameObject` with the `TemporalVisualEffect`, and `bool UpdateTime()` function responsible for updating the timer on the icon, and checking the time left.

#### One Time Temporal Effect

The `OneTimeTemporalEffect` is an abstract class extending the `TemporalEffect` base class that is used for temporal effects that are applied once at the beginning. It overrides the `bool Update()` function so that the effect is canceled after the effect ends.

There are several effects of this type implemented, namely:

- `ApplyEffectOnWeapon` – sets effects that the weapon should apply on collision and changes its material
- `ArmorBoost` – applies a permanent armor boost
- `AttackBoost` – applies a temporary attack boost
- `SpellPowerBoost` – applies a temporary spell power boost

### 5.5.3 Repeated Temporal Effect

The `RepeatedTemporalEffect` is an abstract class extending the `TemporalEffect` base class that is used for temporal effects applied over time. It overrides the `bool Update()` function so that the effect is applied at a given frequency.

There are several effects of this type currently implemented in the game:

- `Burn` – deals damage over time
- `Poison` – deals damage over time
- `RegenerateHealth` – heals over time

### 5.5.4 Projectile Skills

The `ProjectileSkill` is a skill effect used for spawning projectiles that apply a `List<ISkillEffect> effects` on collision. It contains a `Projectile prefab` it should spawn. All the logic of the projectile is then implemented in the `Projectile` class.

The `Projectile` class implements a `IEnumerator ExecuteAction()` function that handles the animation of the projectile slowly spawning, and afterward applies force to the projectile in the given direction. The class also resolves the collision of the projectile and applies the given skill effects on collision with the given target.

## 5.6 Target

Targeting of skill effects is handled using the subclasses of the `ITarget` abstract class. `ITarget` extends the `ScriptableObject` so we can easily create targeting objects that are then assigned to the skill effects.

The `ITarget` class contains `TargetingData` that can be specified by the skill effect during initialization of the `ITarget` object.

The class then has two virtual functions, `List<Unit> GetTargetUnits()` and `List<Vector2> GetTargetPositions()` which return `null` by default, and always just one of these two functions is overridden. The skill effect then calls from the `Apply(Unit unit)` function `ApplyOnTargets(unit, targets)` or `ApplyOnPositions(unit, positions)` based on the return type from the `ITarget` object.

Currently, there are four types of targets implemented:

- `SelfTarget` – returns the owner unit of the skill
- `PlayerCursorTarget` – if the owner unit is an enemy, it returns the position of the player, otherwise it returns the position of the mouse cursor
- `AlliesInRangeTargets` – returns all allies in an area, the area can be either a circle or a cone
- `EnemiesInRangeTargets` – returns all enemies in an area, the area can be either a circle or a cone

# 6. Attacks

Dungeon Crawlers are medium to fast-paced games where both the enemies, as well as the player, have the capability to perform various kinds of offenses. These include Ranged attacks and Melee attacks. Other than these, the units may possess certain special abilities like healing and summoning smaller, less powerful units. In Dungeon of Chaos, the component `IAttack` is responsible for all the attacks performed by the units. The `IAttack` class has been designed in a way that allows for multiple attacks for both `Enemy` and `Player` units.

## 6.1 Base Attack

All the different attack types are derived from the base class `IAttack`. The `IAttack` class is `abstract` and includes the virtual methods `Attack()` as well as the `abstract coroutine IEnumerator StartAttackAnimation()`. To minimize the use of art assets, the attack animations have been coded, and each attack has its own version of the attack. The `IAttack` component is attached to the `GameObject` with the `Weapon` component. There can be more than one `IAttack` component attached to the `Unit` based on the nature of the `Unit` combat. The `Unit` can call the `Attack()` method from the selected `IAttack` component to launch the said attack. A `ScriptableObject` of type `AttackConfiguration` is provided to the `IAttack` component to configure the parameters of the attack such as Damage, Range, Cooldown, Indicator, etc.
An attack `coroutine` mostly consists of three parts. The first part is to prepare the weapon which includes enabling the collider if the attack requires it, enabling the weapon trail, and setting its position as well as orientation.
The second part of the attack includes the entire animation sequence (Listing 7.1) of the weapon corresponding to the type of attack as well as checking for the hits in case of region-based attacks (stomp, smash).

```
    // Sweep
    var startRotation = Weapon.Asset.localRotation;
    float startRotationZ =
       Weapon.Asset.localRotation.eulerAngles.z < 180 ?
       Weapon.Asset.localRotation.eulerAngles.z :
       Weapon.Asset.localRotation.eulerAngles.z - 360f;

    time = 0;
    while (time <= 1) {
        time += (Time.deltaTime /
           attackAnimationDurationOneWay);
        float currentPos = Tweens.EaseOutExponential(time);
        Weapon.transform.localPosition =
           Vector3.Slerp(endPosUpAdjusted - startPos,
           endPosdownAdjusted - startPos, currentPos) +
           startPos;
        yield return null;
    }

    // Backward
    time = 0;
    while (time <= 1) {
        time += (Time.deltaTime /
           attackAnimationDurationOneWay);
        Weapon.transform.localPosition =
           Vector3.Lerp(endPosdownAdjusted, startPos,
           time);
        yield return null;
    }
```

Listing 6.1: Code Snippet for the Swing Attack Animation

The third part of the attack `coroutine` resets the weapon to the state before the attack animation begins.

## 6.2 Melee Attack

Melee attacks are the kind of attacks with very low range and varying accuracy, and damage values. There are four different types of Melee attacks present in the game - Smash, Stomp, Swing, and Dash. All these attacks derive from the `Abstract` class `MeleeAttacks` which derives from `IAttack`.

### 6.2.1 Smash

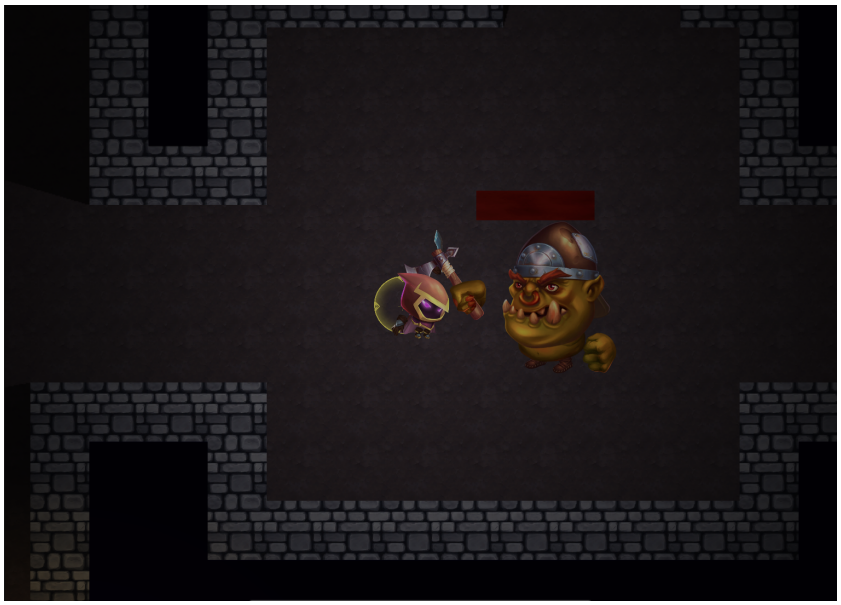

Figure 6.1: Smash Attack Indicator

It is a Melee Attack of the type `SmashAttack`. Smash attack has medium-low range and high damage. The area of attack is a circular region with a very small area but with very high damage. The area of attack as well as other parameters can be configured using the
`SmashAttackConfiguration` (`AttackConfiguration`)
SO. Figure 6.1 shows a yellow circular region, indicating where the smash attack of the enemy would land.

### 6.2.2 Stomp

It is a Melee Attack of the type `StompAttack`. Stomp attack has two different ranges (see figure 6.2) The first area of impact has a circular region of very high damage but a small area. The second area of impact has a circular region of very low damage but a significantly larger area. Both the smaller as well, the larger area of attack and other parameters can be configured using the `StompAttackConfiguration` (`AttackConfiguration`) SO.

### 6.2.3 Swing

It is a Melee Attack of the type `SwingAttack`. The range of Swing attacks is relatively low, and it is usually low-damage wielding. The area of attack resembles the sector of the circle whose angle, radius, and other parameters can be configured using the `SwingAttackConfiguration` (`AttackConfiguration`)

Figure 6.2: Stomp Attack Indicator

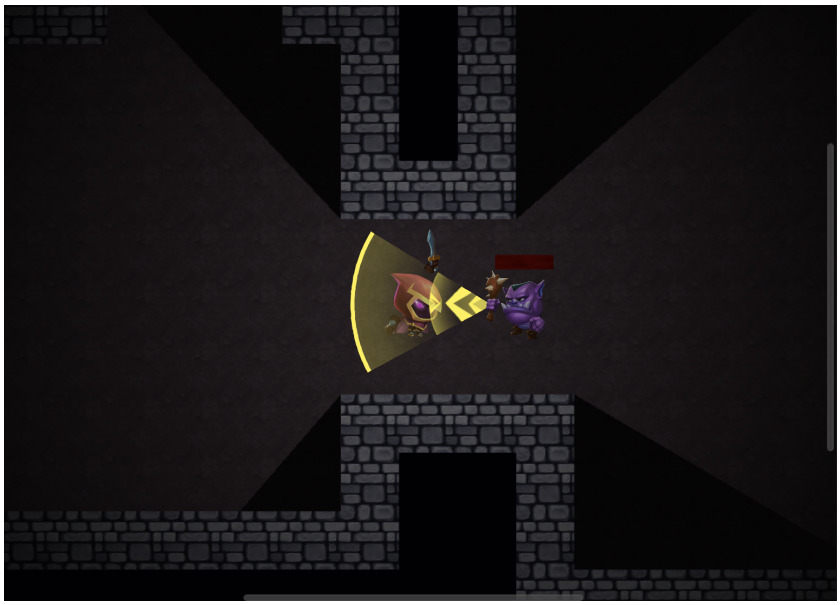

Figure 6.3: Swing Attack Indicator

SO. Figure 6.3 shows the area that the swing attack of that particular enemy covers.

### 6.2.4 Dash

It is a Melee Attack of the type `DashAttack`. During the Dash Attack, the Unit charges at the opponent at high speed and collides to inflict damage. The range of Dash attack is low to medium, and it is usually medium damage wielding. The range and other parameters can be configured using the `DashAttackConfiguration` (`AttackConfiguration`) SO. In figure **??** the in-

Figure 6.4: Dash Attack Indicator

dicator of the dash attack can be seen as a straight line indicating the direction of the dash.

## 6.3 Range Attack

Range attacks are the kind of attacks that usually have a high range and varying accuracy, and damage values. Range attacks have a compulsory component `IProjectile` provided to its `AttackConfiguration` SO. The IProjectile controls the nature of the projectile that is spawned by the Ranged Attack. There are two different types of Range attacks present in the game - Basic and Burst. All these attacks come from the class `RangedAttacks` which derives from `IAttack`.

### 6.3.1 Basic Range Attack

It is the default `RangedAttack` responsible for the basic ranged attacks. The range is usually high, but the accuracy varies based on the nature of `IProjectile`. The range and other parameters can be configured using the `RangedAttackConfiguration` ( `AttackConfiguration` ) SO.

### 6.3.2 Burst Range Attack

Burst Attack is a form of a Ranged Attack where the Unit fires multiple projectiles at once (see figure 6.5). Burst Attacks is controlled by the Class `BurstRangedAttack`, which is derived from `RangedAttack`. The burst and other parameters can be configured using the `BurstRangedAttackConfiguration` ( `AttackConfiguration` ) SO.

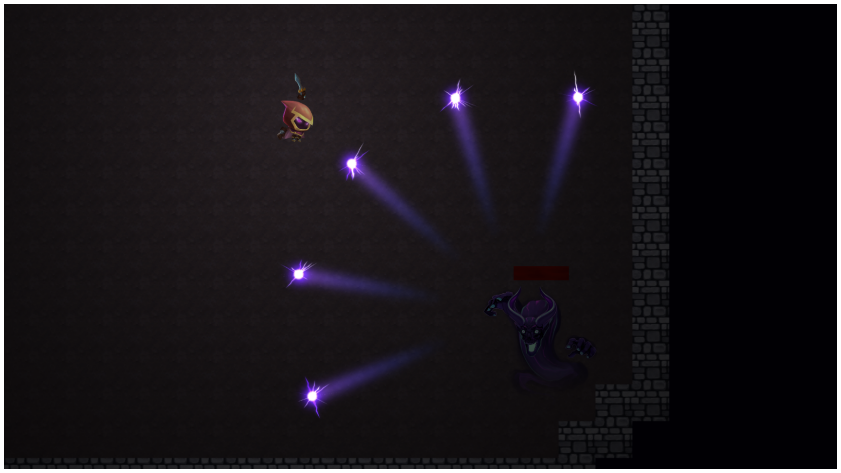

Figure 6.5: Burst Projectiles

### 6.3.3 Projectile

The behavior of the Projectile is governed by the Class `IProjectile`. It is an Abstract Class with public method `Launch(Vector2 direction)` that is called from the `Attack()` method of `RangedAttack`. The `IProjectile` extends two Classes - Basic and Follow. The `GameObject` projectile to which `IProjectile` is attached is configured using the `ProjectileConfiguration` SO. This is used to configure the speed, burst, etc of the Projectile.

#### Basic Projectile

This is the default Projectile that extends from the `IProjectile` class. It moves in a fixed direction that is provided to it when the `Launch(Vector2 direction)` is called.

#### Follow Projectile

This is the second type of Projectile that extends from the `IProjectile` class. The behavior of this projectile imitates a homing missile and follows the player around (see figure 6.6). It launches in the direction that is provided to it when the `Launch (Vector2 direction)` is called but changes its path based on the movement of the player. The variable `maxSteerForce` determines how quickly it aligns with the direction of the position of the player. This variable can be configured using the `FollowProjectileConfiguration` (`ProjectileConfiguration`) SO.

Figure 6.6: Homing Projectile

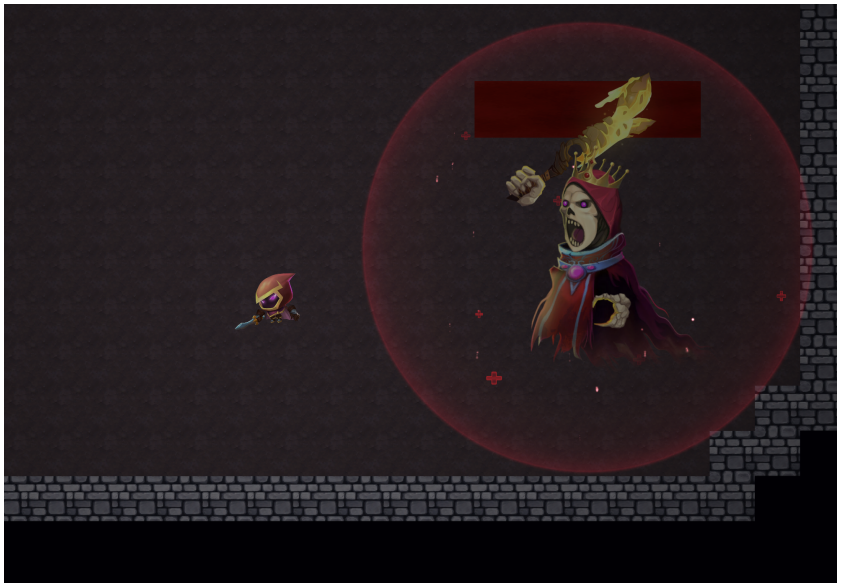

Figure 6.7: Heal Attack Indicator

## 6.4 Special Attack - Heal

`HealAttack` is a kind of attack that falls under the special attack category. It allows the Unit to heal itself by a certain `healAmount` as well as other units of a similar kind within in the `healRadius`. It also extends from `IAttack`. Figure 6.7 shows the heal indicator as a circular region with a certain heal radius.

## 6.5 Special Attack - Summon

`SummonAttack` is the second attack that falls under the special attack category. Summon allows a `Unit` to summon other less powerful `Unit`s (Minion) of its kind within the `spawnRadius`. It also extends from `IAttack`. Figure 6.8 shows minions spawning inside the summon region.

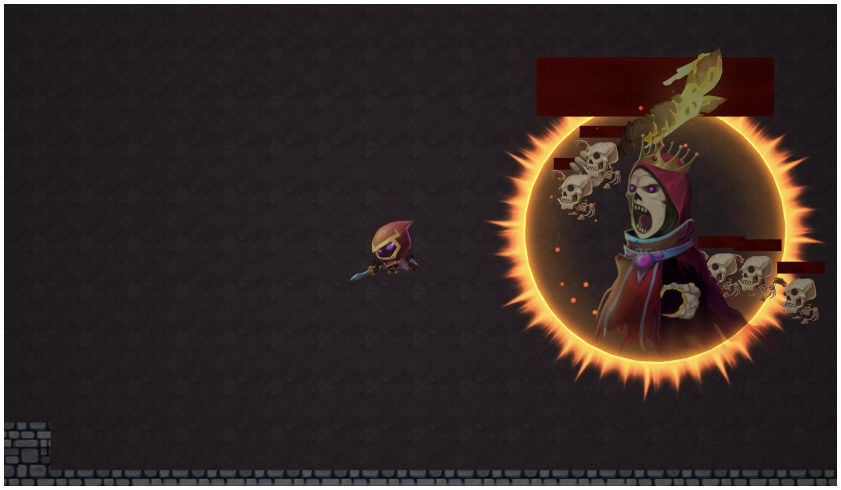

Figure 6.8: Summon Attack Indicator

## 6.6 Indicators

Indicators act as a heads-up for the player to know when an enemy is about to launch a certain type of attack. They are provided as a `Prefab` to the `AttackConfiguration` SO of an `IAttack` under the variable IIndicator indicator. It also requires an SO of the type `IndicatorConfiguration` that configures the various parameters of the Indicator prefab as well as the Indicator spawn animation. `IIndicator` has a public method `Use()` that is called from within the `Attack()` method of `IAttack()` before the attack animation starts. `IIndicator` also has an `abstract` `coroutine` `IEnumerator` `ShowIndicator()` that controls how the Indicator spawns and various kinds of scale and visual `tweens` that can be performed on it. Each `IAttack` has an indicator corresponding to it.

# 7. Dungeon

This chapter deals with the map and its components. We will deal with each of them in the following sections. Many interactive elements make the gameplay enjoyable. Each of them individually is quite simple.

## 7.1 Hierarchical Wave Function Collapse

The map is generated using the hierarchical wave function collapse algorithm. The algorithm is experimental and available at https://github.com/fileho/Hierarchical-Wave-Function-Collapse with more information.

For us, the algorithm itself is not very important since it serves more like a black box. We only need to know how to use it to generate new levels and export them to be usable in the game. This is described in the design documentation.

The generation is done at design time to ensure their quality. Hand-picked outputs are used for each dungeon map. We have also made slight manual post-processing for some dungeons.

This algorithm is an external reference for us. Therefore, we will not describe how it works. The algorithm outputs a `GameObject[,]` of specified width and height. It is a very flexible but very ineffective representation. This size of a standard map is $120 \times 110$ tiles. It would create over 13000 `GameObject` that would slow down our performance dramatically.

Therefore, we create several `Tilemap` for more effective representation. The rest of this section deals with the conversion and interactive map elements.

## 7.2 Tilemap Generator

The purpose of `TilemapGenerator` is to extract objects from the generated map and to fill them into several `Tilemap`.

Tilemaps:

- `Tilemap ground` — stores ground tiles
- `Tilemap walls` — stores walls, has attached `TilemapCollider2D` to prevent movements of units over it

- `Tilemap rocks` — stores rocks and other movement-blocking elements like tombstones, has attached `TilemapCollider2D` to prevent movements of units over it
- `Transform objects` — stores objects that provide run-time behavior; those are mainly torches

That `Tilemap` are stored in a `struct` called `Tilemaps`.

We need to convert each `GameObject` into a `TileBase` and place it to the corresponding `Tilemap`. Luckily, we can modify the `GameObjects` that we get out of the map generation. Therefore, we can attach a custom `ExportTile` component of each `GameObject`. It allows us to specify a `TileBase` and a `Tilemap` in `Tilemaps`.

Then we can run the `TilemapGenerator`. It loops over the `GameObject[,]`. It extracts `List<ExportTile>` for each position. The reason for it is simple. Some `GameObject` can be nested. It can contain a child that should also be extracted. For example, if we have a ground and a decoration in a single position. At this point, placing those tiles into proper `Tilemap` is straightforward.

The `virtutal void Place(Tilemaps maps, Vector3Int pos)` function is defined in the `ExportTile` class. This allows us to derive from this class and override this function. It is done by an `ExportObject` class used for torches. Torches need a custom script attached. Therefore, we will represent each torch as an individual `GameObjects`.

Now we know how the map is created. However, this was only the first step toward a fully working dungeon.

## 7.3 Shadows

Shadows are the essential feature for immersion inside the dungeon. The player's character casts light. We can use shadows to create a dark atmosphere where the player does not know what to expect behind the corner.

The rendering is done using the Universal Render Pipeline. This pipeline provides support for 2D lights and shadows. However, those features are still experimental.

The shadows are created using a `ShadowCaster2D`. We can adjust the shape of the shadow in the inspector. However, it has a significant flaw. It does not provide any API to modify the shape by code. The shape is represented as `Vector3[]`, and it is stored as a `private` variable called `m_ShapePath`. The solution is simple. We can use a reflection to access this field. We can see the vital portion of the code in listing 7.1.

```
private void ChangeShape(Vector3[] newShape) {
    ShadowCaster2D shadowCaster2D =
       GetComponent<ShadowCaster2D>();
    BindingFlags accessFlagsPrivate =
       BindingFlags.NonPublic | BindingFlags.Instance;
    FieldInfo shapePathField =
       typeof(ShadowCaster2D).GetField("m_ShapePath",
       accessFlagsPrivate);
    shapePathField.SetValue(shadowCaster2D, newShape);
}
```

Listing 7.1: Modify of `ShadowCaster2D`

Shadows are cast only for the player's light source. Shadows from different lights would add much less and could hurt our performance. Also, it would prevent us from placing the torches on the walls since they would be inside the shadow caster.

We want all shadows to behave as one. One shadow should not cast shadows over a second since it looks unpleasant in 2D. This functionality is achieved via a `CompositeShadowCaster2D`, which takes no parameters. It is placed in the object containing all different `ShadowCaster2D` to blend their shadows. In our case, it is the *ShadowGenerator* prefab, and all shadow casters must be placed under it.

The game also has very weak ambient light that slightly lights up the dungeon. However, this light gets even weaker with each subsequent dungeon making the atmosphere darker.

### 7.3.1 Shadow Generator

We need to cover the map with shadow casters. It is infeasible to do it manually. Therefore, it is done automatically by the `ShadowGenerator` script. It places an object with a shadow over each contiguous wall segment. It is done via a *BakeShadows* button attached to its component.

It finds a position that still needs to be covered. Then it walks over its edge. The always-turn-right approach it used. This ensures we walk along the whole edge back to the initial position. A special case handles dead ends — it places all the corner control points and turns. Then we must move to the other side of the wall and repeat the process. Finally, we can create a single shape out of it. All of this is implemented in `TrackEdges()`. The whole process is repeated until there are no more areas to cover. It is crucial to add control points only at

the ends of straight lines of walls. Otherwise, we could waste a bunch of memory and performance since we would get a much more complex shape.

It bakes shadow in the editor at design time. Therefore, it does not add any run-time cost. The generating is done by pressing a *BakeShadow* button that is added to the `ShadowGeneration` component. When the generation is done, the scene is saved a reloaded automatically. The correct shadow casting works only after the scene reload — it is probably caused by the internal behavior of `ShadowCaster2D` and Unity.

### 7.3.2 Rocks and Tombstones

There are also rocks in the dungeon. Those rocks also block the movement of the player. Therefore, they need to cast shadows to make it intuitive. Rocks are stand-alone objects and have complex shapes. We manually created one `ShadowCaster2D` for each rock. Then we can instantiate an object with the correct shadow over each rock.

There are also tombstones. However, they behave exactly the same way as rocks.

## 7.4 Minimap

The minimap is very useful for navigation inside the dungeon. It is achieved using a second camera which is zoomed out to cover the whole dungeon. However, the camera does not render all layers. It is set in `CullingMask`.

Rendered Layers:

- *Player* — renders the character
- *PlayerAttack* — renders the player's weapon
- *BothMaps* — objects that are set to appear in both the main map and the minimap; those are, for example, walls
- *MinimapOnly* — objects that should be rendered only in the minimap

The minimap does not render the ambient light. Therefore, it is totally dark almost everywhere. We render the character so the player can know his relative position inside the dungeon. It also renders all visited checkpoints. We need to save all visited checkpoints, and then we can set their layer to *BothMaps*. Since the checkpoint also emits light, we can see its close neighborhood.

## 7.5 Objects

The following sections describe different dungeon objects. Each object has an impact on the gameplay.

### 7.5.1 Checkpoints

Figure 7.1: Checkpoint

We can see the checkpoint in figure 7.1. The checkpoint is the most crucial map element. It allows the player to save the game and opens the *CharacterUISheet* for character progression.

The checkpoint contains a collider with `IsTrigger = true` to determine if the player is near. If he does, a *Canvas* appears with the *F* key. When the player presses the *F* key, the *CharacterUISheet* opens. The game is saved when the player closes it, and the scene is reloaded to reset and respawn enemies.

When the player rests at the checkpoint, the checkpoint is saved. From now on, this checkpoint will be visible in the minimap. It also sets a respawn point to the last rest area.

Checkpoint emits light using a `Light2D`. It also emits smoke using a particle system with noise.

### 7.5.2 Fog

The fog separates the boss room from the rest of the dungeon. It is directional, and the player can go through it only in one direction. And it is pretty sophisticated. We can see the fog in figure 7.2.

The most crucial part of the fog is its visual effect. It is achieved via a `ParticleSystem`. Those particles are partially transparent and use noise to move constantly.

Then there is a collider that blocks the movement of the player. Another collider with `IsTrigger = true` determines if the player stands before the fog.

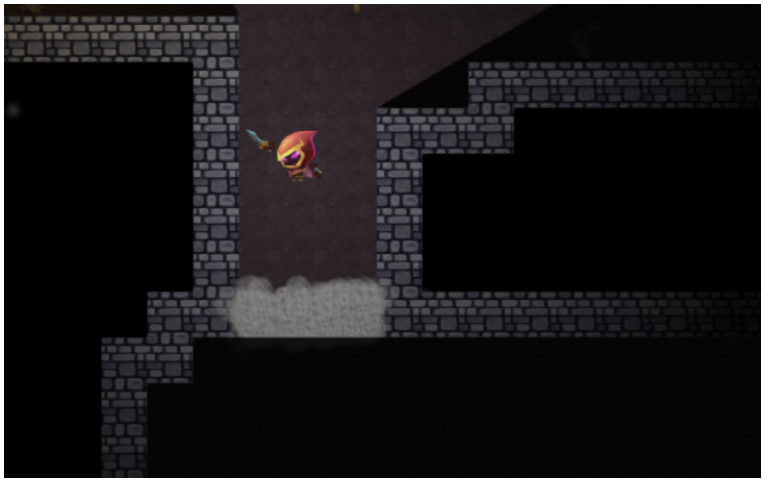

Figure 7.2: Fog

If he does, a *Canvas* appears with the *F* key. When the player presses the *F* key, an animation will move him through the fog. We can rotate the fog into all four cardinal directions, and the *Cavnas* will always properly adjust its position and rotation.

The `ShadowCaster2D` is an essential part of the fog. Otherwise, we would be able to see into the boss room. We use a custom script that sets its shape to be pixel-perfect. It is called when the *BakeShadows* button is pressed on the `ShadowGenerator`.

### 7.5.3 Map Fragments

The map fragment can reveal part of the minimap to the player. Again the same approach is used to show a Canvas with the *F* button. It needs to be saved the same way as checkpoints.

Implementation-wise, the minimap renders only specific layers. Besides that, it ignores ambient light. Therefore, most of the map is black. We can reveal parts of it using lights. In particular, each map fragment uses a sizeable square light to reveal a significant portion of the map to the player. The layer of those lights is set to *MinimapOnly*, so they are not visible in the main camera.

The light must set the `BlendStyle` to `BlendStyle3`. This style uses a `RenderTextureScale = 0.01f`. Typically, this causes horrible artifacts. However, here it does not, and it nicely blends two overlapping light sources. Otherwise, the blend is unpleasant.

### 7.5.4 Chests

Chests are used to reward the player for exploration. Again we show a Canvas with the $F$ button. When opened, it drops loot. It is $EXPEssence$ for now, but it can drop anything else.

Since it drops loot, we must save whether the chest was already open. In that case, we do not want to drop any loot and immediately set the chest as open.

A chest also contains a `ShadowCaster2D`. The shadow caster has a different shape for the opened and closed version of the chest. However, we can active the proper `GameObject` at runtime, and everything works fine.

### 7.5.5 Crates

Crates are very similar to chests, but crates are destroyable. The primary difference is that the crate does not drop loot when destroyed. Therefore, we can always respawn crates and do not need to save them.

Undestroyed crates also cast shadows using a `ShadowCaster2D`. We can keep its shape the same since creates are rectangular. Destroying a crate removes its shadow correctly, so we do not have any extra work.

### 7.5.6 Torches

Torches are placed by the generation. Their purpose is to give some life to the dungeon. By default, the torch is off. However, when the player gets close, it lights up slowly and emits particles. The torch will slowly turn off when the player gets too far away.

We have two types of torches — a regular torch and a blue torch. Blue torches are used deeper in the dungeon since those cold colors support the dark atmosphere.

# 8. Saves

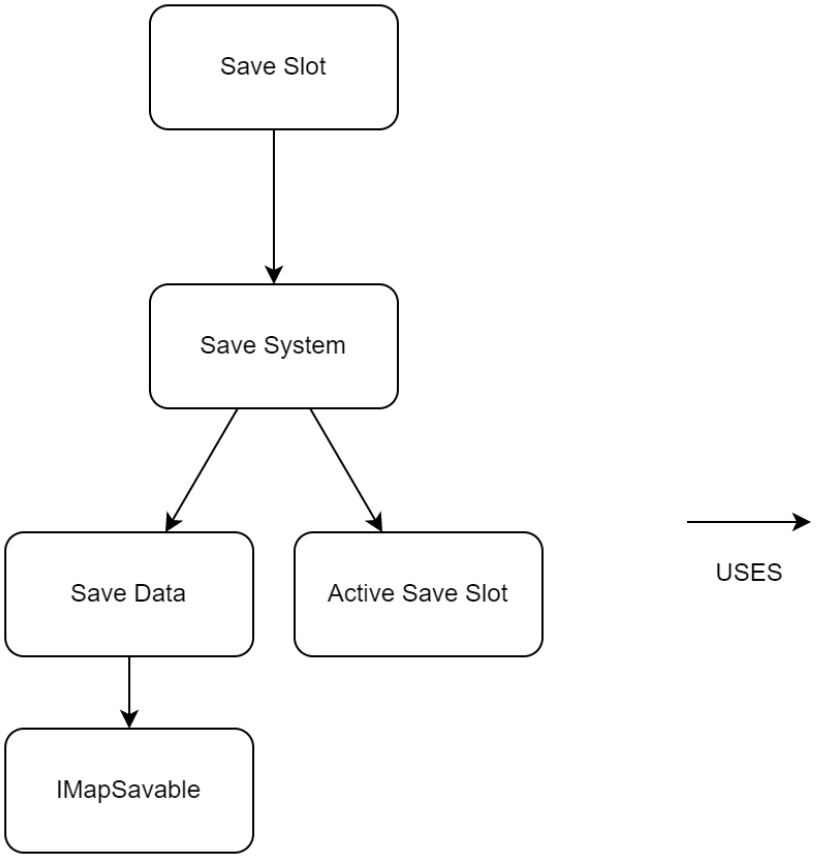


Figure 8.1: Save System

This system is responsible for saving and loading all the required data. Its overview is shown in figure 8.1. This system is unique since it needs to communicate back and forth with many different systems. However, we have decided to cover it later since it is a stable system that should not change much.

The save the done only when the player enters a Checkpoint. At that point, we need to save the position of the player, active, state of the dungeon (visited checkpoints, collected map fragments...), and data of the character progression like stats and skills. Then all this data gets serialized and written in a binary format to a single file for each save. The player can have up to three active saves at once.

The data are loaded when the scene is loaded. It happens either when the player respawns after they die. The scene is also loaded when the player leaves the checkpoint to respawn all enemies. At that point, we load the saved file, deserialize it, and restore the state of all systems mentioned above.

## 8.1 Save System

The `SaveSystem` is the most important script for saving and loading data. It writes data to files for persistent storage between gameplay sessions. In fact, it happens much more often since we are reloading the scene after each death. The data are saved when leaving a checkpoint.

The save system allows us to have multiple saves at the same time. The path to the current active save is stored in a different file. It is stored using the `ActiveSaveSlot` script.

Public API:

- `public void SaveProgress()` — writes the current holding data into a file
- `public void Load()` — loads data from a file; if the file can not be found, a new save with default values is created
- `public void LevelComplete()` — level is complete; increments the current dungeon and removes all saved data about the dungeon
- `public SaveData GetSavedData(int saveSlotIndex)` — used visualize the save information for the player
- `public void SetSaveSlot(int index)` — sets a path for the `ActiveSaveSlot` script and writes it to file. Used to load a save in the main menu

Writing data to files uses the `BinaryFormatter` to serialize and deserialize the data. The `BinaryFormatter` is outdated. However, it provides us with a very simple way of serializing and deserializing binary data.

The save system also contains three public properties — the `SaveData`, the `DungeonData`, and the `TutorialData`. Those fields can be modified externally. Furthermore, those fields are used when the `SaveProgress()` is called. Otherwise, the caller would need to know all the data that should be saved. Those data come from many different systems, so maintaining them would be pretty unpleasant. The rest of the data required to be saved are extracted from the `Character`.

The save system also has a simple custom editor. It allows us to delete the current save directly. It is a handy feature for testing. There is also a button *Save/Remove All Saves* at the top of the editor window.

The save system only handles the storing and loading of the data. The logic of when it should happen is handled elsewhere. All of the functions are called only by a `GameController`, which handles the high-level logic.

## 8.2 Save Data

The `SaveData` struct contains all data that will the written to a file.

`SaveData` fields:

- `public Vector3s characterPosition` — the position of the Character; `Vector3s` is a serializable version similar to the `Vector3`
- `public SavedStats savedStats` — current players combat values
- `public SavedSkillSystem savedSkillSystem` — unlocked and equipped skills
- `public DungeonData dungeonData` — all data savable from the dungeon; objects implementing the `IMapSavable` interface
- `public DateTime timestamp` — timestamp of the save; important mainly for the user
- `public TutorialData tutorialData` — the state of the tutorial were already shown

All of the fields contain raw data. Each system that needs to be saved should provide a similar `struct` that contains only data that needs to be saved. It does not make sense to save a whole system with many parameters that cannot change dynamically. Besides that, the system should also provide two functions.

Functions for systems:

- `public T Save()` — T represents the raw data type; export all the fields that need to be saved
- `public Load(T data)` — loads the state from the provided data

### 8.2.1 Map Savable Interface

The `IMapSavable` interface serves for saving dungeon objects. For these objects, we save a state of whether they were already used or visited.

`IMapSavable` functions:

- `void SetUniqueId(int uid)` — set a unique id to the object; it is required so we can identify the object
- `int GetUniqueId()` — gets the unique id
- `void Load()` — what should happen when the object is saved
- `Object GetAttachedComponent()` — should return the attached component; it is required so we can set it as dirty in the inspector, and it can be saved

Those unique ids are assigned at design-time with a custom editor-only script *Dungeon/Assign Ids*. In the `DungeonData`, we need to save an `int` as the index current dungeon and a `List<int> mapElements` — unique ids of objects that were activated and implementing the `IMapSavable`.

Data that must be saved from the remaining systems are discussed in appropriate chapters.

## 8.3 Active Save Slot

We have mentioned that we support multiple save slots. Therefore, we need to know which save is active. It is stored in the `ActiveSaveSlot`. We only need to know the index of the current save. Therefore, this class only stores a single `int` at the dist. Again, it used the `BinaryFormatter` to serialize and deserialize the data.

## 8.4 Save Slots

The `SaveSlots` script is not a part of the main gameplay scenes. It is used for selecting a save in the main menu. It needs to present the relevant data to the player. Therefore, it uses the `SaveData GetSavedData(int saveSlotIndex)` function in the `SaveSystem` public API. We get all the information from it and display relevant parts of this data to the player.

Besides that, the `SaveSlots` also handles loading a save from the menu. When a save is selected, it sets its index to the `ActiveSaveSlot`, so the `SaveSystem` knows which save to use. After it, it can load the appropriate scene specified in the save.

# 9. Sound System

This system is responsible for playing all sound effects.

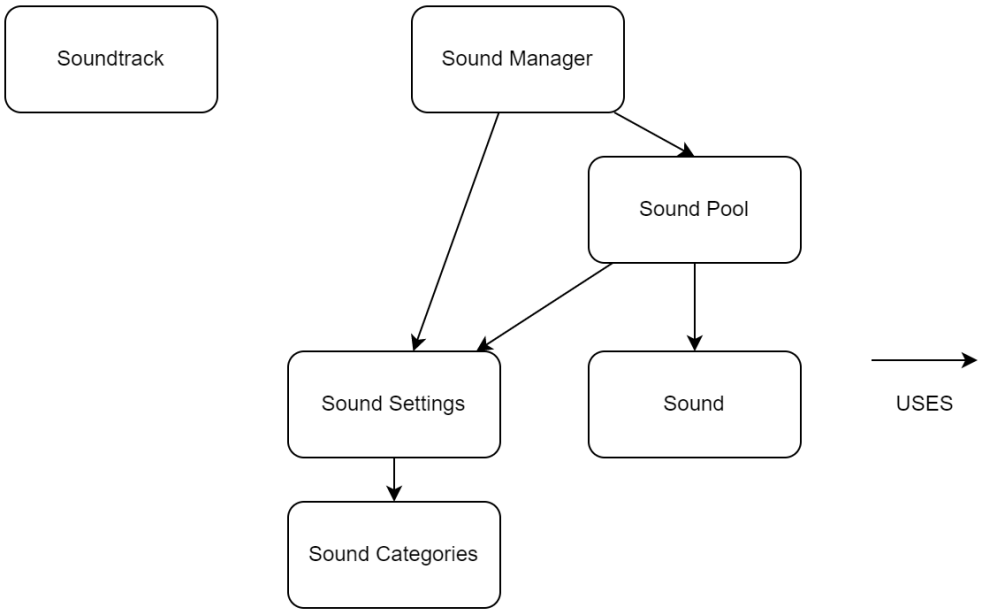


Figure 9.1: Sound System

The structure of the whole sound system is shown in figure 9.1. We will describe them in more detail in the following sections. Since sounds are a crucial part of each game, the sound system must be part of each scene.

## 9.1 Soundtrack

The playing of soundtracks is implemented in the `Soundtrack` script. It is independent of the rest of the sound system.

`Soundtrack` needs to meet these properties:

- Do not stop when a different scene is loaded
- There needs to be exactly one in each scene at the time

Both of those features are implemented with a single simple function 9.1.

```
private void Awake() {
    var soundtracks = FindObjectsOfType<Soundtrack>();
    if (soundtracks.Count > 1) {
        // destroy new object if some already exist
        destroy(gameObject);
        return;
    }
    // do not destroy this object on scene loads
    DontDestroyOnLoad(gameObject);
}
```

Listing 9.1: Soundtrack Invariant

The public API is straightforward. It contains two functions.

Public API:

- `public void PlayBossMusic(int level)` — start playing the boss music for the current level if it is not playing already
- `public void StopBossMusic()` — stop playing the boss music, play regular music again

Input fields:

- `[SerializeField] private List<AudioSource> music;` — all song playable normally
- `[SerializeField] private List<AudioSource> bossMusic;` — all songs playable during a boss fight

The Soundtrack is internally using a single `AudioSource` to play the music. It is a state machine that keeps a `bool` whether it should play standard or boss music. The second parameter determining the state is an `int` with the current song index. When the current music ends, it will play a new one based on its state. It can also quickly transition to another `AudioClip` — start playing the boss music. We need the Soundtrack to be responsive when the situation changes.

## 9.2 Sound Manger

The `SoundManager` is the main class responsible for the playing of sounds.

Public API:

- `public void PlaySound(SoundSettings sound)`
  — finds given `AudioClip`, available `AudioSource` and plays the sound
- `public SoundData PlaySoundLooping(SoundSettings sound)`
  — finds given `AudioClip`, available `AudioSource` and plays the loop sound; returns a handle
- `public void StopLoopingSound(SoundData sound)`
  — stops the looping sound
- `public void UpdateLoopingSound(SoundData sound, float volume)`
  — updates a volume of the looping sound

The `PlaySound(SoundSettings)` is the most critical function of this chapter. All sounds are played using this function, and every script that wants to play a sound needs to call it. There is a Prefab called *SoundSystem* that we should put into each scene.

For the looping sounds, there is a set of functions. Playing the looping sound returns a read-only handle. The handle is required to stop or update the sound. The caller is responsible for stopping the looping sound when it is no longer necessary.

Sounds are played using a sound pool. The sound manager contains several `SoundPool` — one for each `SoundCategory`.

### 9.2.1 Sound Categories

Sounds are not stored in the script that wants to play the given sound effect. All sounds are stored directly in the sound system. Sound Categories are options for specifying which sound we want to play.

Sound Categories are only several enums. Each category must be defined in `enum SoundCategory`.

`SoundCategory` values:

- `Looping`
- `EnemyAmbients`
- `Attack`
- `Skill`
- `Ui`

- `Items`
- `Death`
- `TakeDamage`

Then for each of the values, we have another `enum` with the given sound. This solution is not type-safe since we must specify the given sound effect somewhere else. However, it is designed to be simple to work with for game designers. Specifying the sound effect should be as easy as possible in the inspector.

### 9.2.2 Sound

The `Sound` is a simple class used to specify a given sound effect.

Input fields:

- `[SerializeField] private AudioClip audioClip;` — the sound to be played
- `[SerializeField] [Range(0f,1f)] private float volume = 1;` — the default volume
- `[SerializeField] [Range(0.5f, 1.5f)] private float pitch = 1;` — the default pitch

It is helpful to have fields for default volume and pitch, so we do not need to ensure that all sound effects have good volume externally.

### 9.2.3 Sound Pool

The sounds are played using the sound pool approach. We have many available `AudioSource`. Furthermore, if we want to play a sound, we must find an empty `AudioSource` and use it. This way, one `AudioSource` can be used for any sound effect.

Input fields:

- `[SerializeField] private int poolSize = 30;` — the number of `AudioSource` we can use
- `[SerializeField] private List<Sound> sounds;` — all sounds playable with this sound pool

The one `SoundPool` should match one `enum` of `SoundCategories`. It is unpleasant since sounds need to be specified in the same other fields of the given `enum`. There is no simple way to check whether the order is correct because there is no link between the `Sound` and the field of the `enum`. Adding a new sound effect to the end of the list and as a last element of the enum is always safe.

We could use `string` instead, but it would be even worse for the game designers.

Public API:

- `public AudioSource FindEmptyAudioSource(float priority)` — finds an available `AudioSource` from the pool; if none is available, it replaces the sound with the lowest priority
- `public Sound GetSoundAtIndex(int index)` — returns the `Sound` on the given index

The `FindEmptyAudioSource(float priority)` function stores an index of the last returned `AudioSource`. When called, it starts the search from that index. Since this `AudioSource` was available, some of the next ones will likely be available too. The complexity is still $\mathcal{O}(n)$. However, it usually succeeded very fast. The same approach is common in particle systems.

We also have a special sound pool for looping sound. It is `SoundPoolLooping`, but the idea behind it is the same.

### 9.2.4 Sound Settings

The caller must implement the `SoundSettings` class to specify a sound and its properties.

Input fields:

- `[SerializeField] private SoundCategory soundCategory;` — the category of the sound effect
- `[SerializeField] private int sound;` — the index of the sound in a given category; it is int since it can originate from different enums and all enums are convertible to int
- `[SerializeField] private float volume = 1;`
- `[SerializeField] private float pitch = 1;`

- `[SerializeField] private float priority = 0;`

The sound itself is represented as `int`. We can convert between the `int` and any value from any `enum`.

### 9.2.5 Sounds Editor

We have already mentioned that we want to make the selection of the given sound effect as simple as possible. Therefore, we have implemented a custom `PropertyDrawer` called `SoundsEditor`. Since the sounds are played from many different scripts, we have to use the custom `PropertyDrawer` instead of the custom `Editor`.

It draws normally the `volume`, `pitch`, `priority`, and `soundCategory` fields. The reason why we need this script is to draw the remaining `sound` field. It looks at a value of the `soundCategory` and draws the appropriate `enum` field. Then it casts the chosen value to an `int` and stores it in the `sound` field. The primary part of the solution is shown in listing 9.2. Firstly, we must create the correct type of the `enum` with the correct value. Then we need to use the `EditorGUI.EnumPopup` since we do not have a `SerializedProperty` that we could use directly. Finally, we can store the resulting value in the appropriate field.

We chose to solve these cases manually. It might be possible to do it via templates and only specify the type for each case. However, the low flexibility of generic types in *C#* makes it unnecessarily complicated.

```
public override void OnGUI(Rect position,
   SerializedProperty property, GUIContent label) {
    // draw rest of fields skipped for simplicity
    // calculate the position of the field
    var soundRect = calculatePos();
    // Draw fields - pass GUIContent.none to each so they
       are drawn without labels
    EditorGUI.PropertyField(categoryRect, soundType,
       GUIContent.none);
    int index = soundType.enumValueIndex;
    switch (index) {
    case 0: {
        SoundCategories.Ambient s =
           (SoundCategories.Ambient)soundIndex.intValue;
        soundIndex.intValue =
           (int)(SoundCategories.Ambient)
           EditorGUI.EnumPopup(soundRect, s);
        break;
    }
    case 1: {
        SoundCategories.FootSteps s =
           (SoundCategories.FootSteps)soundIndex.intValue;
        soundIndex.intValue =
           (int)(SoundCategories.FootSteps)
           EditorGUI.EnumPopup(soundRect, s);
        break;
    }
    case 2: {
        SoundCategories.Attack s = (SoundCategories.Attack)
            soundIndex.intValue;
        soundIndex.intValue =
           (int)(SoundCategories.Attack)
           EditorGUI.EnumPopup(soundRect, s);
        break;
    }
    // remaining cases solved similarly
}
```

Listing 9.2: Sounds Editor — enums

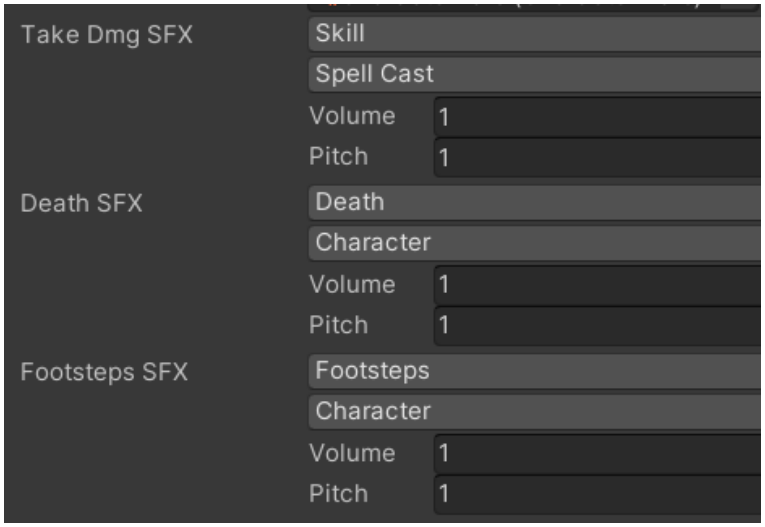


Figure 9.2: Sound Settings in Inspector

Figure 9.2 shows how it looks in the inspector. The second `emun` automatically changes based on the value in the first `enum`. Both $DeathSFX$ and $FootstepsSFX$ have the selected sound called $Character$. However, each of them refers to a different sound effect since both of them are from different categories.

Splitting those effects into categories is crucial. Otherwise, it would be difficult to find the proper sound effect.

### 9.2.6 Adding New Sounds

This system is relatively easily extendable. Adding new sounds is very simple. An `AudioClip` needs to be added at the end of the appropriate `SoundPool`. Furthermore, a value for it needs to be added to the given `enum`, so the sound effect can be specified in the inspector.

Adding a whole new category of sounds is more tricky. Firstly, it needs to be added to the `SoundCategory`. Also, a new `enum` must be created to specify new sounds. Secondly, a new pool must be added to the `SoundManager`, including all new sound effects. Finally, the appropriate field must be added to the `SoundsEditor`. However, this should not be common that a whole new category is required.

# 10. Tutorials

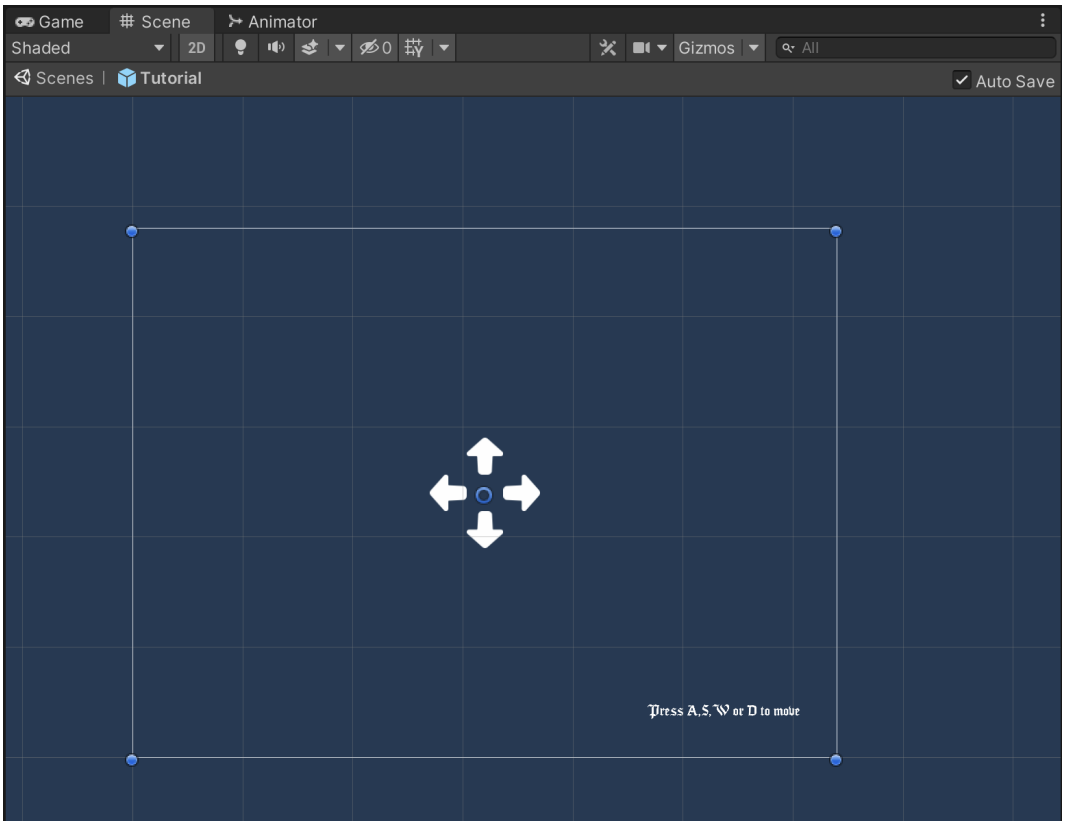


Figure 10.1: Basic Movement Tutorial Screen

Although the game's controls are pretty straightforward, and there are instructions available on the settings screen, a separate tutorial has been included to make the user onboarding smoother and easier. The tutorial is a mix of interactive + static, which is easier to implement and, at the same time, intuitive enough to make the user understand. The tutorial operates a simple FSM at its core. The several tutorial screens are the states that transition to the next screens(states).

Each tutorial has an area. The tutorial is triggered when the player enters it. Then, the time is slightly slowed down, and new instructions are shown to the player. This way, we can show the tutorial whenever we need it. Tutorials are also saved to be shown only once in each run unless the player dies.

We have the following tutorials to the core game concepts:

1. Movement
2. Dash
3. Attack and Stamina

4. Enemy attacks, indicators, and dodging
5. Checkpoint
6. Minimap
7. Boss

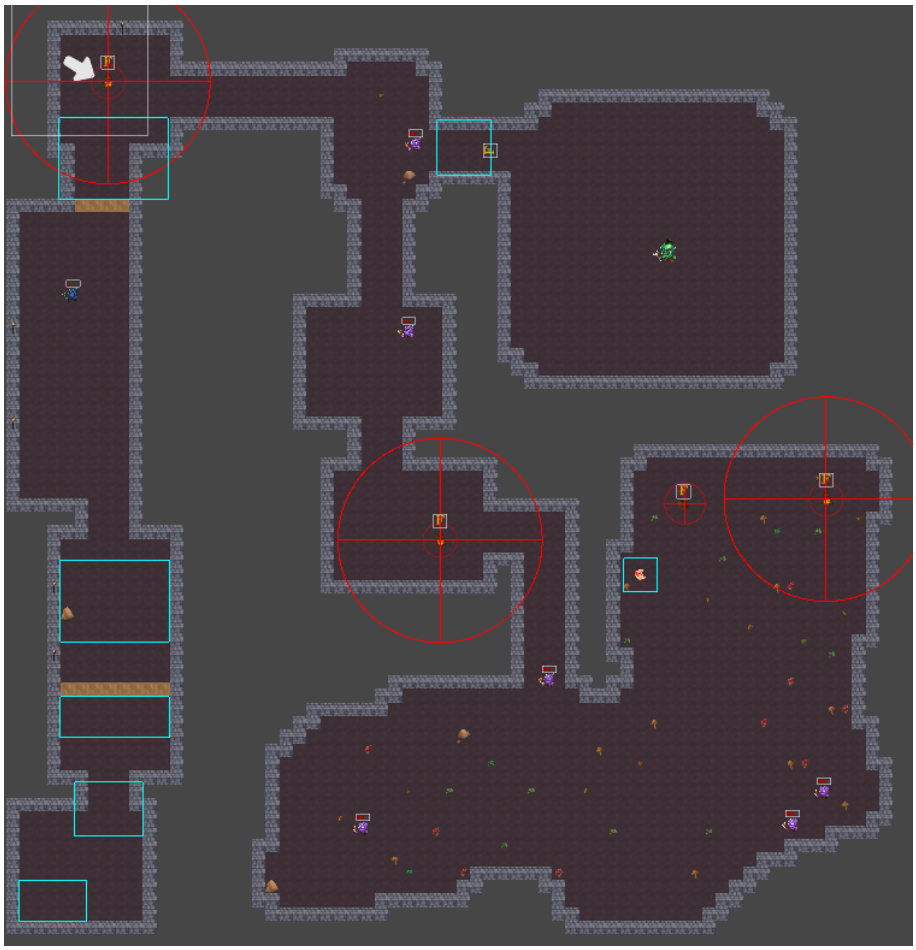

Figure 10.2: Tutorial triggers

Figure 10.2 shows the tutorial dungeon. The blue boxes are the trigger areas for each tutorial. This way, we can delay tutorials to a point, where they are required and do not totally destroy the flow of the game at the start.

## 10.1 Tutorial Prefab

The `Prefab` *Tutorial* is the parent `GameObject` that includes all the screens meant to be part of the tutorial. It also includes the `AnimatorController` component and the `TutorialManager` attached to it. Each tutorial screen has its own `Animation` object that is passed to the `AnimatorController` when its

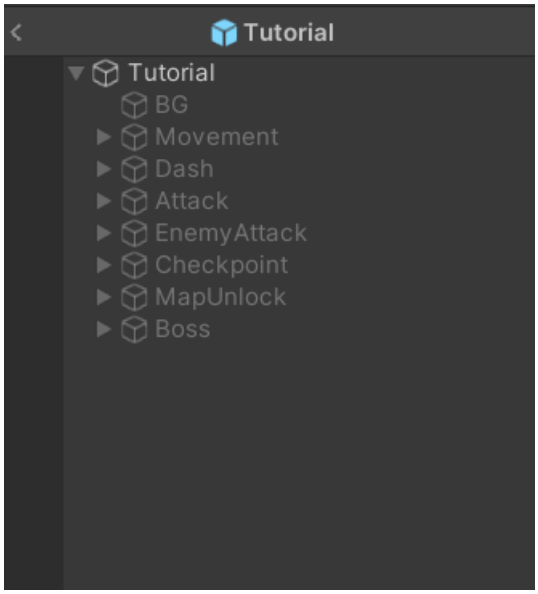


Figure 10.3: Hierarchy of the Tutorial Prefab

corresponding state is active. The tutorial screens parented to `Prefab` *Tutorial* are in the order that corresponds to their order in the `Enum TutorialState`.

# A. Attachments

## A.1 Design Documentation

The GDD is located in a separate file in order to increase the readability of both documents. It can be found at the following link: GDD.

## A.2 Contributions

### A.2.1 Audio

All soundtracks were composed by AIVA (Artificial Intelligence Virtual Artist): `https://www.aiva.ai` and copyright to those tracks are owned by AIVA.

Sound effects were downloaded from `freesound.org`, `zapsplat.com`, `https://mixkit.co/free-stock-music/` and from `https://sonniss.com/gameaudiogdc`. The complete list of SFX used is listed below:

- horror_creature_goblin_monster_short_throaty_grunt – Zapsplat
- human_male_man__angry_growl_grunt_short_02 – Zapsplat
- zapsplat_horror_creature_goblin_monster_short_grunt_growl_unhappy_003
- zapsplat_sport_cricket_bat_swing
- zapsplat_impacts_body_hit_punch_or_kick_002_90383
- Gravel Footsteps - Studio 23 Ltd (GDC Game Audio Bundle)
- Gamemaster Audio (GDC Game Audio Bundle)
- zapsplat_foley_stick_branch_medium_sized_swing_whoosh_swoosh_through_air_001_90352
- zapsplat_impacts_body_hit_baseball_bat_hard_whack_crack_squelchy_bloody_005_44151
- zapsplat_fantasy_magic_dark_spell_cast_whoosh_fast_002_77590
- zapsplat_impact_body_slam_hit_against_metal_surface_hard_004_43755
- human_male_man__angry_growl_grunt_short_03 – Zapsplat
- horror_creature_goblin_monster_short_grunt_growl_unhappy_001 – Zapsplat

- hit_injured_5 – Sounds Great – Orc Creature Library (GDC 2020 Game Audio Bundle)
- zapsplat_warfare_axe_or_other_heavy_weapon_swing_005_25155
- zapsplat_horror_axe_chop_impact_person_fleshy_wound_guts_001_14319
- troll_monster_taunt03 – GameMaster Audio
- troll_monster_hurt_pain_01 – GameMaster Audio
- troll_monster_growl_long_05 – GameMaster Audio
- warfare_sword_or_other_weapon_swing – Zapsplat
- impacts_body_hit_baseball_bat_hard_crack – Zapsplat
- warfare_weapon_heavy_axe_swing – Zapsplat
- Giant Footsteps – Blastwave
- `https://freesound.org/s/660683/`
- zapsplat_horror_monster_demon_screech_002_23962
- `https://freesound.org/s/636089/`
- zapsplat_impacts_punch_clothing_body_hit_with_fast_short_whoosh_001_81253
- zapsplat_impacts_body_hit_punch_or_kick_012_90393
- zapsplat_horror_ghostly_dark_breath_long_reverb_005_78751
- zapsplat_animals_snake_hiss_short_14693
- PM_Ghostly+Breath+Whoosh+binaural+AMBEO – Zapsplat
- zapsplat_foley_clothing_shirt_or_tshirt_swing_punch_movement_001_67066
- zapsplat_horror_knife_slice_slash_skin_fleshy_gore_004_13216
- smartsound_FOLEY_CHAIN_MAIL_Armor_Movement_01
- soundbits_ScreamsShouts2_Monster_Roar_Growl_179
- MeleeSwingPackNormalSwings – David Dumais (GDC Game Audio)
- MeleeSwingPackHighSwings – David Dumais (GDC Game Audio)
- zapsplat_horror_ghostly_dark_breath_long_reverb_001_78747

- animals_alligator_hiss_002 – Zapsplat
- zapsplat_horror_ghostly_dark_breath_long_reverb_006_78752
- warfare_medieval_katana_sword_swing_002 – Zapsplat
- zapsplat_impacts_swoosh_into_metal_thud_hit_002_43406
- zapsplat_warfare_sword_swipe_hit_body_impact_hard_squelch_20829
- zapsplat_horror_ghostly_breath_long_reverb_011_83403
- Skeleton Bodyfall – QuickSounds
- zapsplat_sound_design_whoosh_slow_dark_heavy_88759
- Whooshes and Impacts – Saro Sahihi, Sound Bits (GDC Game Audio Bundle)
- Magic Spell Cast Short Air Whistling – David Dumais (GDC Game Audio Bundle)
- Dark Spell Life Tap – Sound Spark LLC (GDC Game Audio)
- Bluezone_BC0269_creature_wood_attack_impact_dust_006
- zapsplat_foley_concrete_block_heavy_short_movement_scrape_001_58267
- zapsplat_foley_large_stone_small_rock_drop_roll_fall_on_dirt_debris_dusty_soil_83230
- MeleeSwingsPack_96khz_Mono_DesignedSwings12
- Bluezone_BC0269_creature_wood_footstep_medium_004
- tspt_chaotic_electric_sizzle_loop_017
- PM_OF_Electro_Magnetic_One_Shots_5
- MAGIC ENERGY Texture, Complex, Particles, Zap, Sparks, Short-Circuit, High, Sweetener 03
- CreatureSnarl – Audiomeal (GDC Game Audio Bundle)
- WHOOSH Ball Crackle Small 02 – Smart Sound FX
- Blastwave_FX_BodyFallSnow_S011HO.32
- mixkit-little-devil-laughing-413

- Bluezone_BC0261_mountain_dweller_creature_orc_grunt_003
- Imp03 – Erdie (freesounds) https://freesound.org/s/31303/
- Flamethrower Three Bursts - QuickSounds.com
- Large Demon Hurt Roar 4 - QuickSounds.com
- MAGIC_SPELL_Flame_03_mono – Imphenzia
- smartsound_FOLEY_ARMOR_Plate_Armor_Movement_Rustle_01
- magic spell – kostas17 (freesounds.org)
- Sword Unsheathe 1 – Quicksounds
- The Sound Pack Tree
- Skyrim Healing – Garyq (Freesounds.org)
- technology_earphone_cable_spin_swish_through_air
- musical_heavenly_choir_003
- zapsplat_household_book_hard_back_pick_up_from_hard_ground_002_54191
- spa_fire_burning_firepit – Silver Platter Audio
- Inspector J – GDC GA
- Ambient Portal Hum – Shiversmedia – Freesound
- little_robot_sound_factory_fantasy_Ambience_Cave_00

### A.2.2 Art

Most of the game art was done by our good friend Arya Harshwardhan. He has been such a crucial part of this project. He has worked really hard to make sure that the artwork was in sync with the theme, and style of the game. We appreciate the time he has taken to understand the requirements, and the patience to accommodate the feedback, and make sure the assets were delivered in time.

### A.2.3 Packages

Assets from the Unity Asset Store have also been used to enhance the visuals of the game. They helped us make the game more appealing. All the third party packages used in the game are listed below:

- A* Pathfinding Project Pro
- All In 1 Sprite Shader
- Doozy UI Manager
- Epic Toon FX