SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Iris Stare

GACHA GAME DEVELOPMENT IN GODOT

UNDERGRADUATE THESIS

Varaždin, 2024.

UNIVERSITY OF ZAGREB

FACULTY OF ORGANISATION AND INFORMATICS

V A R A Ž D I N

Iris Stare

Identification number: 0036536833

Course: Informacijski i poslovni sustavi 1.2

GACHA GAME DEVELOPMENT IN GODOT

UNDERGRADUATE THESIS

Mentor:

Doc. dr. sc. Mladen Konecki

Varaždin, September 2024.

Iris Stare

Statement of authenticity

I declare that my undergraduate thesis is the original result of my own work, in the making of which no resources outside of the ones mentioned in the work were used. Ethically appropriate and acceptable methods and techniques were used to create this work.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Summary

This work outlines the creation of a game prototype in the Godot game engine. The prototype aims to create a game that blends the role-playing game genre with the gacha game genre. The main gameplay consists of the player attacking enemies using drawn gestures in order to collect currency which is used to play gacha. The player engages with the gacha aspects to unlock new gestures to use in battle. The work will showcase the game's design and visual elements. Then, the game's architecture will be covered in detail, including the ability system, the scene architecture, the singletons that control the flow of the game, the components which scenes use, and the custom data objects used to represent various elements of the game. At the end, it can be concluded that choosing the role-playing and gacha genres to develop a game is incredibly demanding due to the technical overhead required to implement the bare functionalities and content both genres require. Nevertheless, this kind of project is an invaluable opportunity to learn more about the development of complex games.

Key words: Godot; video game; development; gacha; RPG; gestures; mechanics; systems; architecture

Table of Contents

1. Introduction

The concept of this undergraduate thesis is to create a prototype for a game that combines role-playing game elements with gacha game elements using the Godot game engine. Specific characteristics of both genres have been chosen to represent the genres in this work. The role-playing game (abbreviated as RPG) aspects will be represented by the battle system the game uses. Conversely, the gacha game aspects will be represented by the abilities the player can use in the game. To unlock new abilities to use in battle, the player has to win the abilities' respective units from the gacha system.

The motivation behind developing an RPG with a gacha system comes from the thought that a gacha system offers incredible replay value for the player. The player would be incentivized to engage with the gacha system in order to win stronger abilities to use in battle. Furthermore, the player would wish to keep playing the game until they have collected all of the available gacha units.

Both RPGs and the gacha games are difficult to develop. The difficulty in developing RPGs lies in the incredible number of different systems that interact together. Gacha games, on the other hand, rely on having a large amount of content, as well as creating a gameplay reason for why the player should engage with the gacha game mechanics. In order to undertake both genres, this work focuses on tying together a small number aspects of both genres. The gacha system is simple and features a small number of rewards. The game implements RPG mechanics with its simplified combat system and levelling mechanics.

2. Methodology

2.1. The Godot Game Engine

Godot is a free, open-source, and general-purpose game engine. Starting life as a private in-house engine made by Juan Linitsesky and Ariel Manzur, Godot was released as an open-source project in 2014. [1] Godot supports both 2D and 3D projects and has export options for several different platforms.

One of Godot's unique qualities is its custom scripting language GDScript. GDScript is a high level, dynamically-typed and object-oriented language similar to Python, and has tight integration with the engine. [2] Due to this and the language's ease of use which contributes to speed of development, the game project uses GDScript as its programming language.

The Godot game engine has a particular design philosophy. The engine is made to encourage object-oriented design using *composition*. [3] Composition allows objects in the game to be composed out of other objects, each defining different behaviours that do not need to be aware of each other to function. Inheritance of classes and defining custom classes is supported by the engine as well.

Given this, the Godot Game Engine has a few key concepts that are integral to any project made in the engine, which are the following [4]:

- **Nodes** – Nodes are the building blocks of every Godot project. Nodes are objects that carry different properties, signals and functionalities out of the box. Godot provides several different node types, such as `Node2Ds`, `Node3Ds`, and `Control` nodes for user interface (abbreviated as *UI*) elements. A more concrete example of a node would be a `RigidBody2D` node, which gives the object adjustable physics settings that will automatically be run and applied upon project compilation. Nodes can inherit from other nodes, and game developers can define their own node types within the engine with their own properties and behaviours. Custom scripts can be attached to nodes to make them execute specific behaviour, such as putting a character movement script on a `CharacterBody2D` node to control the character.
- **Scenes** – Scenes can be roughly compared to Unity's concept of *Prefabs*, with added flexibility. Scenes are collections of several different nodes and can serve different purposes. A scene can be as small as a collection of several different nodes that make up a Player scene – for example, a `CharacterBody2D` node as root, with a `CollisionShape2D` for providing collision functionality, and a `Sprite2D` node to provide visuals for the scene. A scene can also be a large collection of

scenes representing an entire level or subset of a game world. Scenes can be nested within other scenes.

- **Scene Tree –** Scenes are contained in a scene tree, similarly to how nodes within scenes are contained in a tree of nodes. Each scene can be treated as its own Scene Tree, though the scene tree usually implies the game's currently-running highest-level scene during runtime.
- **Signals** – Signals are particular to the Godot Game Engine, and allow developers to implement the observer programming pattern [5] project-wide. Each node given by the engine comes with its own set of signals that fire when certain conditions are met. For example, the `Area2D` node has a signal that fires whenever a node of type `CharacterBody2D` enters its collision radius. Developers can define their own custom signals in code, as well as connect other signals to specific nodes.

2.2. Godot Plugins

The Godot Engine has its own *Asset Library*, where developers publish useful plugins that extend the engine's capabilities or provide extra functionalities. Several open-source and free plugins have been used in the development of the game project:

- **Godot Resource Groups** [6] – This plugin allows the loading of resources from folders on runtime. Generally, when a Godot project is compiled, all existing file paths for loading resources or files become encoded, and therefore code which relies on file paths becomes invalid. This plugin allows the easy loading and reading of resources on runtime using file paths within the project even after compilation;
- **Edit Resources as Table** [7] - This plugin makes it easier to see all the resources within a given folder and easily edit some of their properties, speeding up creation of new assets;
- **Aseprite Wizard** [8] – The Aseprite Wizard plugin allows easy integration of *.aseprite* files in the Godot Engine. In practice, it automates the process of assigning animations to `Sprite2D` and `AnimationPlayer` nodes from *.aseprite* files. The plugin provides different *.aseprite* file import options, as well.

2.3. Aseprite

Aseprite [9] is a software primarily meant for creating pixel art assets. It was first released in 2014 and is developed by Igara Studio. Aseprite offers advanced sprite creation and animation creation tools. All game assets have been created using Aseprite.

3. Game Design

3.1. Genre

The game project falls under the *role-playing game* (abbreviated as *RPG*) and *gacha game* genres.

Role-playing games, in the context of video games, have a wide range of definitions and properties. Most commonly, RPGs are defined as games where the player controls a character within a fictional world. RPGs generally have a storyline that players can experience through the completion of quests. Besides that, RPGs can have a variety of different game mechanics and gameplay. Some RPGs make use of a *turn-based system*, where the player can take their time to make decisions at their own pace. Some others use more dynamic, real-time systems such as *action RPGs*. RPGs place great importance on the characters and settings in the games.

Gacha games are a newer genre of video game. The main system of gacha games is a reliance on *gachapon*-inspired systems as part of the core gameplay loop, where a player spends some sort of material or currency to receive random rewards. Gacha games are notorious for their monetization practices and are often compared to the *loot box* system of other games. [9]

This game project incorporates the character, story-telling and combat elements of RPGs with the random reward systems of gacha games. There is no monetization model; instead, a player invests in-game resources to receive new playable units as their gacha rewards.

3.2. Genre Inspirations

3.2.1. Xenoblade Chronicles 2

Xenoblade Chronicles 2 (abbreviated as *XC2*) is the primary inspiration behind the prototype. *Xenoblade Chronicles 2* is an RPG released in 2017 where players can play *gacha* to win living weapons (called *Blades*) to use in the game. Materials to summon new Blades can be gained through quests, with the player getting access to special Blades over the course of the story by completing specific story objectives. XC2 features an incredibly long storyline, taking over 100 hours to finish in some cases, with a rich cast of playable characters and a large cast of Blades. The Xenoblade series of games are also known for their unique lore and world-building. Every game’s playable world being set on top of gigantic titans. [10]

Figure 1: Xenoblade Chronicles 2's cover art (Monolith Soft, 2017., retrieved 2024.)

3.2.2. Honkai: Star Rail

Honkai: Star Rail (often abbreviated as *HSR*) is an RPG and gacha game developed and released by miHoYo in 2023. *HSR* features traditional turn-based combat with a rich cast of characters. As of the time of writing, there’s currently 56 characters in the game that have been or are currently obtainable through the gacha system. Each character has their storyline unlocked through *Companion Quests*, which are designed to make the character more desirable to win in the gacha. Another motivation is the characters’ special abilities in combat. The game has an overarching storyline across several different areas, each with its own distinct gimmick enemies and cast of gacha characters.

Figure 2: Honkai: Star Rail's logo (miHoYo, 2023., retrieved 2024.)

3.3. Story

The story for the game underwent several revisions. Initially, the game was supposed to have a grim-dark atmosphere, accompanied by a story which would incorporate elements like spirits, ghosts and praying at temples. However, as the development went on and the visual aesthetics of the assets changed due to time constraints and interest, a different narrative started to emerge.

Eventually, the story took on a light-hearted tone in order to aid its new aesthetics. Moreover, it was made to give context behind the gameplay loop, tying the story and gameplay together to further incentivize the player.

The story is short and simple. It starts by introducing us to the player character. The player character has a book, but its pages get stolen by some forest animals. The player needs to confront the animals and defeat them in order to get back the tome’s pages. Using the retrieved pages, the player can summon new drawings.

The aspects of defeating the animals and retrieving the pages tie together the main concepts of the game – RPG battles and gacha.

Figure 3: Cutscene screenshot (Original work, 2024.)

3.4. Gameplay

3.4.1. Battle System

One of the core parts of every RPG is its battle system. The game doesn't follow traditional turn-based conventions; instead, the player character's turn is based upon the player's gesture drawing speed, while the enemies use an Active Time Battle system to determine turn order. Active Time Battle*,* abbreviated as ATB, replaces the concept of linear turn queues with internal timers on each participant in the battle, where the battle participant's turn would arrive when the timer runs out, or fills the ATB gauge [18].

To attack, the player draws gestures over the enemies. The gestures trigger their corresponding abilities to affect the targets. The player is able to unlock gestures to use in battle by playing the gacha. After unlocking a gesture, the player can use it in battle as many times as they wish, whenever they wish. The player needs to successfully finish a battle to get awarded with currency to use in the gacha.

The way the ATB system works in the prototype is explained in more detail in section *4.4.3 – ATB Component*.

3.4.2. Gacha System

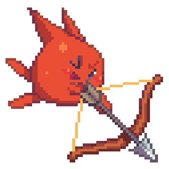
The gacha system in this game is used as a tool to encourage prolonged gameplay with added replay value, where the player would be incentivized to keep playing until they've gotten desirable units, or collected all of the units currently available in the game. The way the gacha system works in the prototype is explained in more detail in section *4.3.7 –Gacha Manager*.

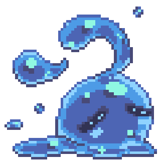
It costs 1 paper – the currency for playing gacha in the game – to do 1 gacha draw.

3.4.3. Gacha Units

Below is a table listing all of the gacha units implemented in the game, along with their rarities. There are 11 different units in total obtainable in the game. All of the unit art is original work.

Table 1: Implemented gacha units

Unit Art	Unit Name	Rarity	Unit Ability
	Archer	Super Rare	Precision Shot
	Bolt	Rare	Thunder Strike
	Catastrophe	Common	Cherry Bomb
	Brick Man	Common	Brick Throw
	Firefly	Common	Lantern Flame
	Lovebug	Common	Renewal

	Lunny	Rare	Moon Shine
	Halley	Super Rare	Meteor Shower
	Speenda	Common	Confusion
	Ichor	Rare	Cleanse Drop
	Venomnion	Rare	Poison Splash

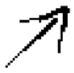
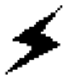
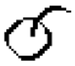
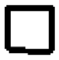
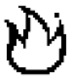
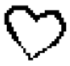
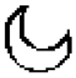
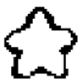
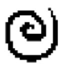
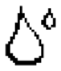
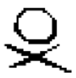
Each gacha unit can be levelled up an infinite number of times. To level up a unit, the player has to pay a gold fee and give up 1 unit of material that the gacha unit needs. The player can obtain additional materials to level up gacha in the event they receive a duplicate of a gacha unit they already have.

When levelled up, the gacha unit's stats increase. The stats rise linearly in relation to the gacha unit's level. The gold fee to level up a unit rises exponentially in relation to the gacha unit's level.

3.4.4. Implemented Gestures

Below is a table listing all of the recognizable gestures in the game, along with the ability that the gesture triggers.

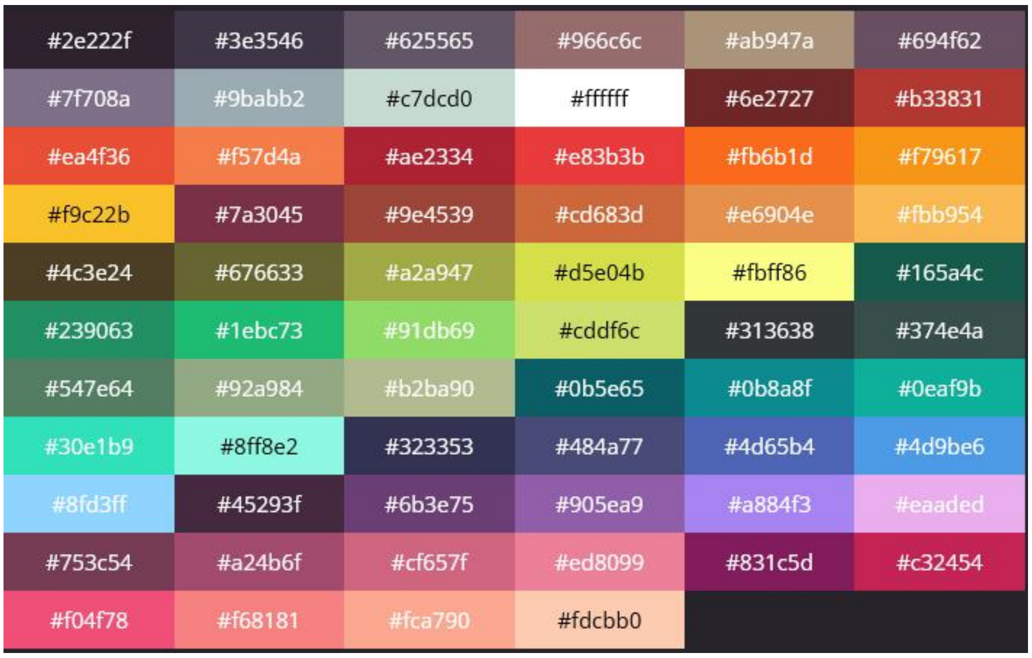
Table 2: List of implemented gestures

Gesture	Gesture Name	Ability	Ability Description
	Arrow	Precision Shot	There is a 50% chance that you will one-shot the enemy.
	Bolt	Thunder Strike	There is a chance that thunder will strike random enemies 1-5 times.
	Bomb	Cherry Bomb	Hits the targeted enemy and its direct neighbours like a bomb.
	Square	Brick Throw	Throw a brick. It will probably land on the enemy.
	Fire	Lantern Flame	Permanently lowers the opponent's maximum HP and burns the enemy over a duration of time.
	Heart	Renewal	Heal yourself by a certain amount.
	Moon	Moon Shine	Induces fear in the opponents. There is a chance for enemies to miss their turn.
	Star	Meteor Shower	Summons a meteor shower. Hits all enemies on the field for a great amount of damage.
	Spiral	Confusion	Induces confusion in the opponents. There is a chance for enemies to attack each other on their turn.
	Waterdrops	Cleanse Drop	Heals the player from any applied status effects.
	Skull	Poison Splash	Poisons the enemies around the target and the target itself. Permanently lowers the target enemy's max HP.

3.5. Visuals

3.5.1. Characters

The game uses pixel art graphics in order to make it easier to produce assets. To unify the visual style, the colour palette *Resurrect 64* made by Kerrie Lake [11] has been used as a guideline in the creation of all assets.



#2e222f	#3e3546	#625565	#966c6c	#ab947a	#694f62
#7f708a	#9babb2	#c7dcd0	#ffffff	#6e2727	#b33831
#ea4f36	#f57d4a	#ae2334	#e83b3b	#fb6b1d	#f79617
#f9c22b	#7a3045	#9e4539	#cd683d	#e6904e	#fbb954
#4c3e24	#676633	#a2a947	#d5e04b	#fbff86	#165a4c
#239063	#1ebc73	#91db69	#cddf6c	#313638	#374e4a
#547e64	#92a984	#b2ba90	#0b5e65	#0b8a8f	#0eaf9b
#30e1b9	#8ff8e2	#323353	#484a77	#4d65b4	#4d9be6
#8fd3ff	#45293f	#6b3e75	#905ea9	#a884f3	#eaaded
#753c54	#a24b6f	#cf657f	#ed8099	#831c5d	#c32454
#f04f78	#f68181	#fca790	#fdcbb0		

Figure 4: *Resurrect 64* palette (Source: Kerrie Lake, 2019.)

The player character's appearance underwent the most changes, going from a tall and lean appearance to a shorter, rounder and more approachable look.

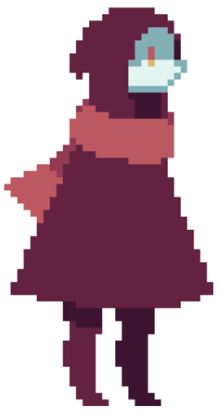


Figure 5: Initial player appearance (Original work, 2024.)

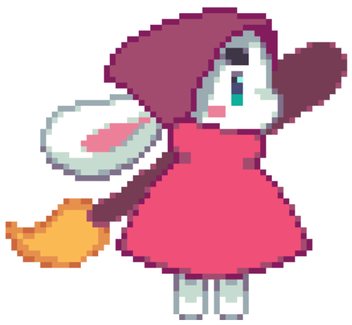


Figure 6: Final player appearance (Original work, 2024.)

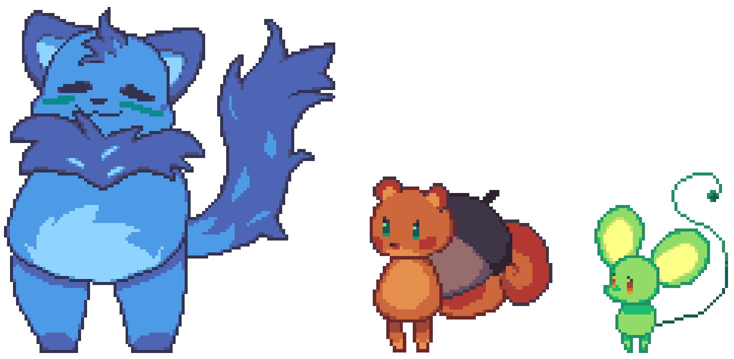


Figure 7: Enemy sprites (Original work, 2024.)

3.5.2. Areas

The game features two playable areas – the house and the forest.

The forest uses a hand-drawn tile set for its foreground environment, while parallax layers serve as a backdrop. The background parallax layers give the forest an illusion of depth and vastness. The forest motif was chosen due to the popular usage of forested and grassy areas as starting zones in most video games (an example being the first starter stages of every 2D *Sonic* game). The forest serves as an area where the player can go and earn more currency to play gacha by defeating enemies, as well as test out newly gotten abilities.

The house is meant to be an area of respite and the protagonist's in-game dwelling. The house would serve as a hub area where the player can heal, play gacha, and manage their units. The writing table (located on the right side of Figure 8.) which functions as a place where the player can summon new units, has been moved from the entrance to the middle of the house (as seen in Figure 9 and Figure 10) in order to encourage the player to walk further into the room and to prevent accidental interactions happening immediately upon entering or exiting the house.

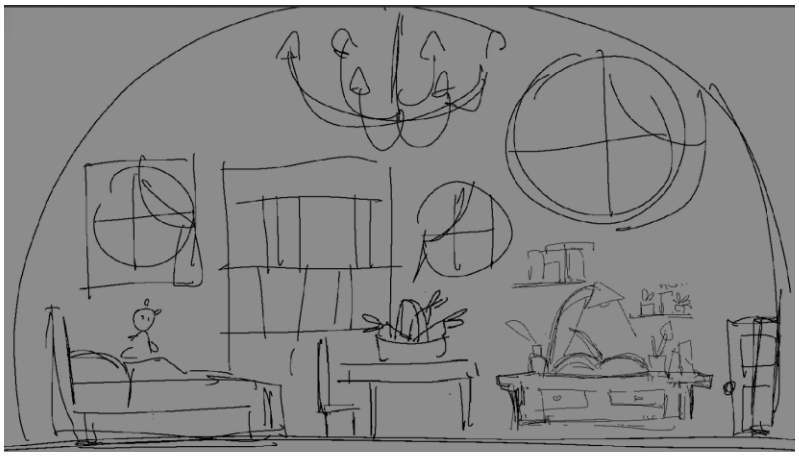


Figure 8: Initial sketch of the player's house (Original work, 2024.)

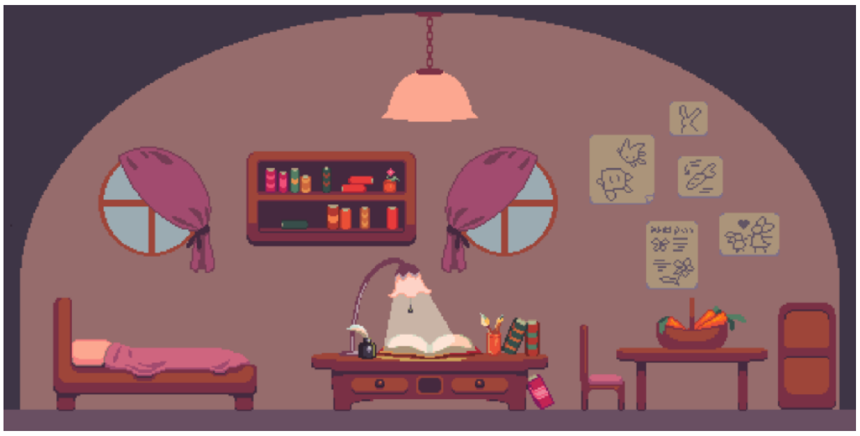


Figure 9: Finished player room asset (Original work, 2024.)

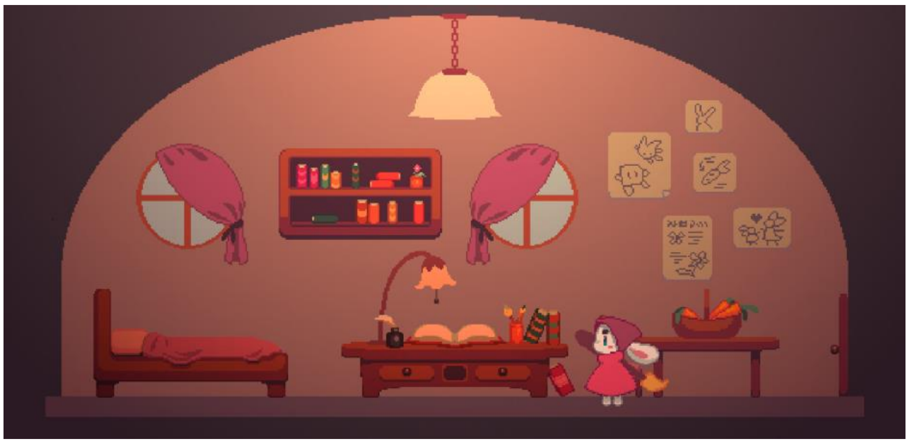


Figure 10: Player house appearance in-game (Original work, 2024.)

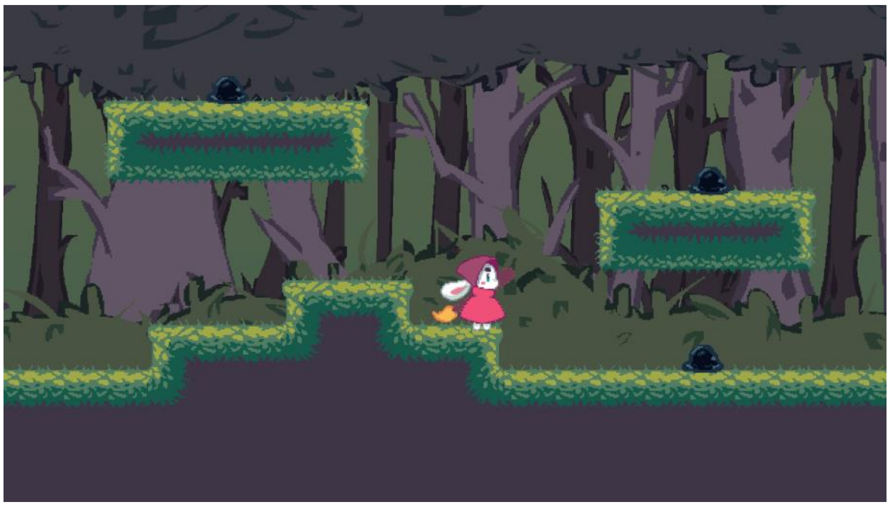


Figure 11: In-game forest area (Original work, 2024.)

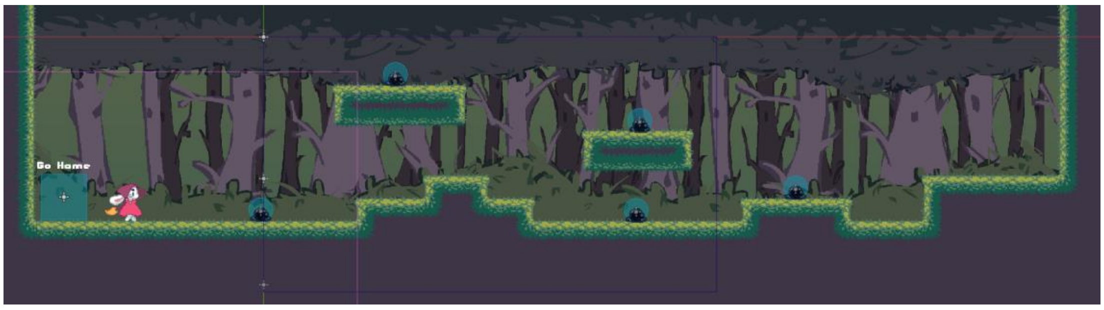


Figure 12: Forest area in-engine (Original work, 2024.)

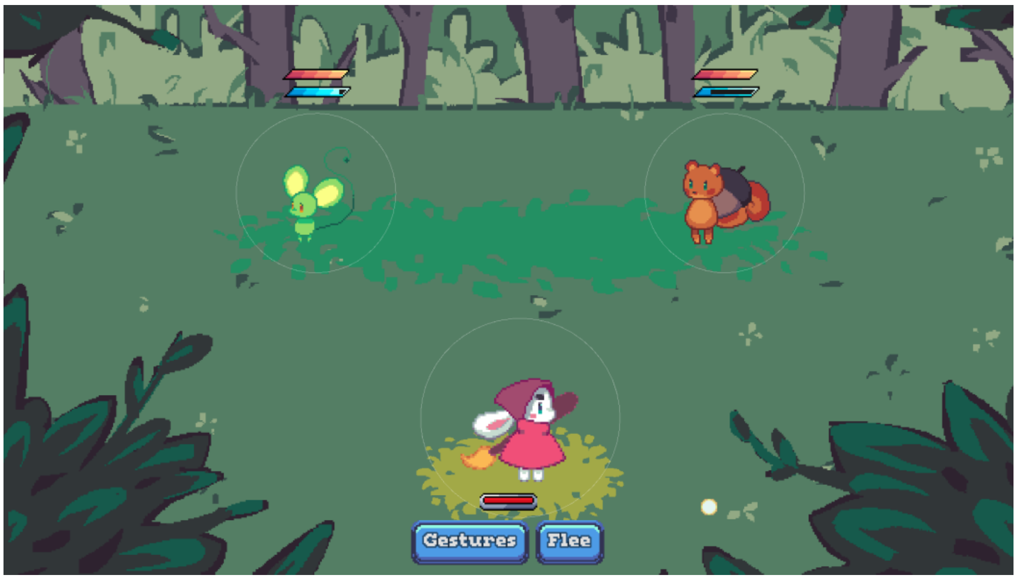


Figure 13: In-game battle scene (Original work, 2024.)

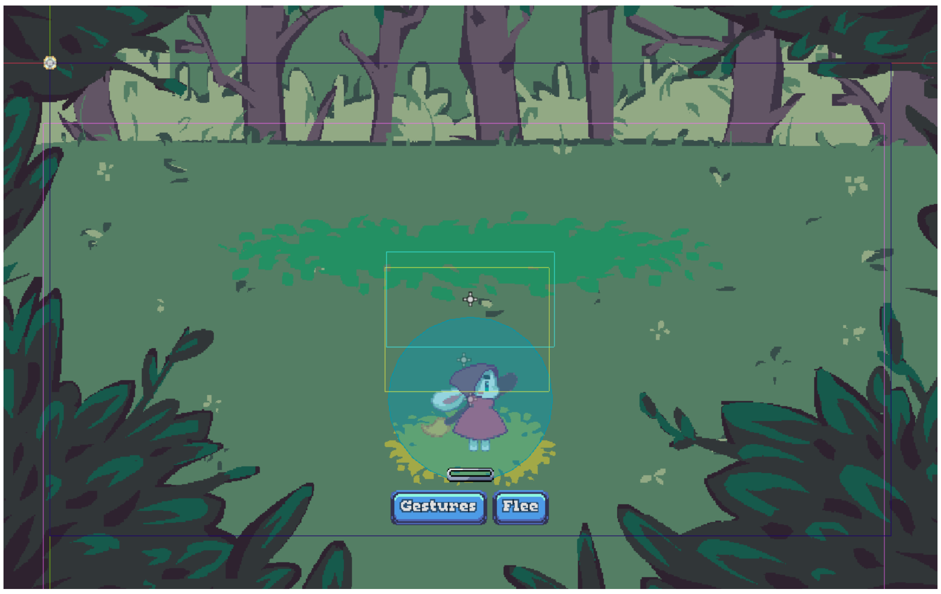


Figure 14: Battle scene in-engine (Original work, 2024.)

3.5.3. Icons and User Interface

The game's user interface (abbreviated as UI) features pixel art and uses a pixel font to display text in order to remain consistent with the character sprites.

Upon starting the game, the start screen is shown. The player has the option of making a new save file, or loading an already existing save file. After selecting a save file, the opening cutscene begins playing.

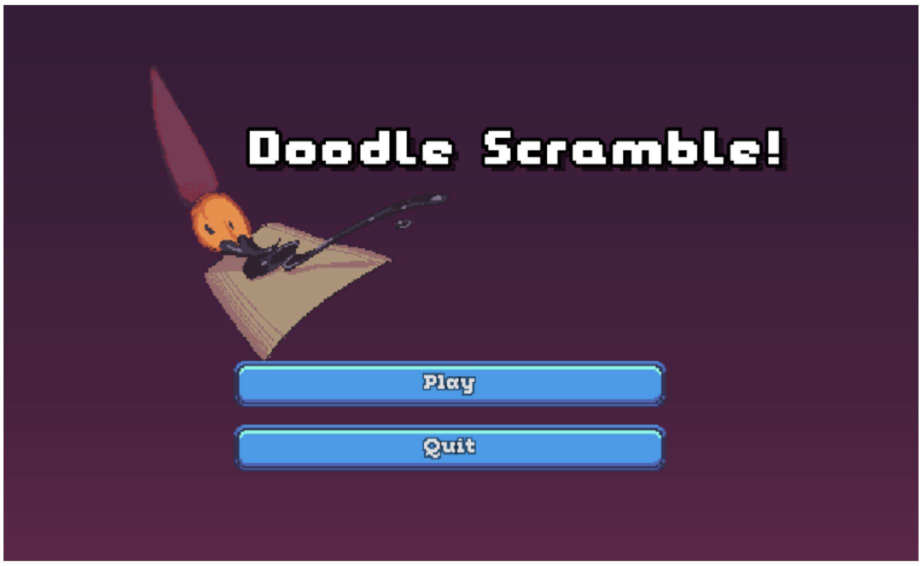


Figure 15: Start screen (Original work, 2024.)

The player, when opening the standard user interface, has access to 3 different tabs with various options: Units, Items, and Menu. Units shows the player a list of the currently acquired units along with more information, Items shows the number of materials the player has in their inventory, and Menu allows the player to save or quit their game.

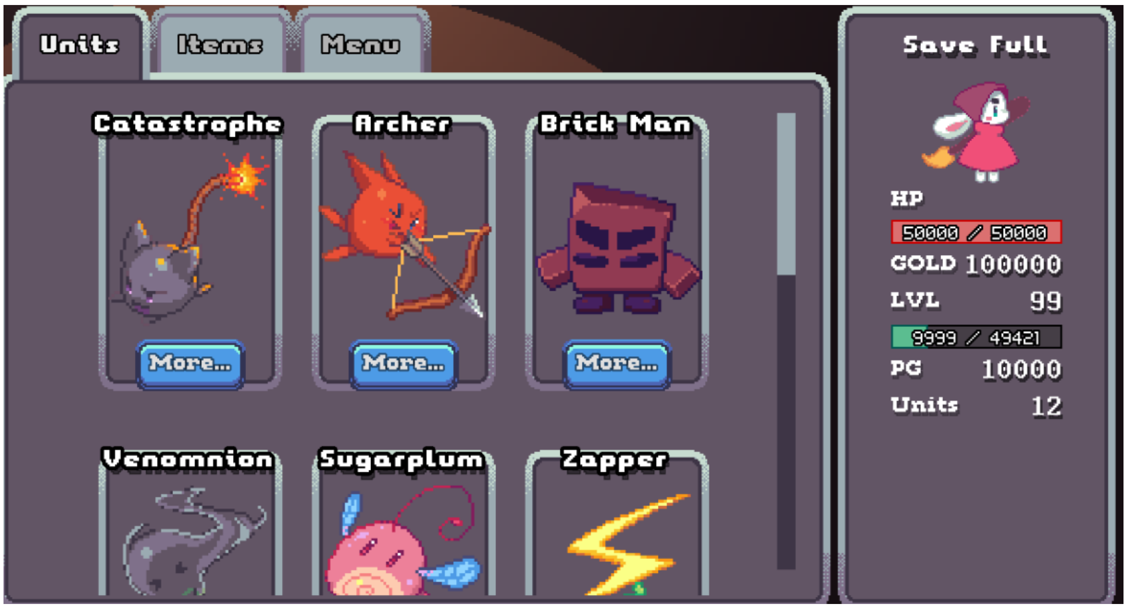


Figure 16: Units tab (Original work, 2024.)

By pressing the “*More…*” button, the player can access the respective unit’s upgrade tab. The player is able to cycle between the units in their possession by pressing the arrow buttons.

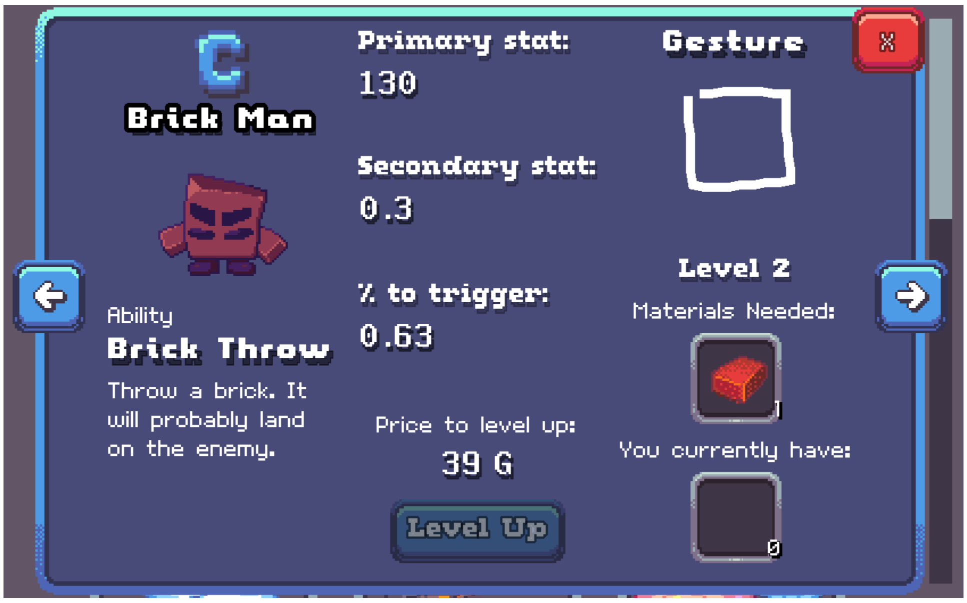


Figure 17: Unit Upgrade window (Original work, 2024.)

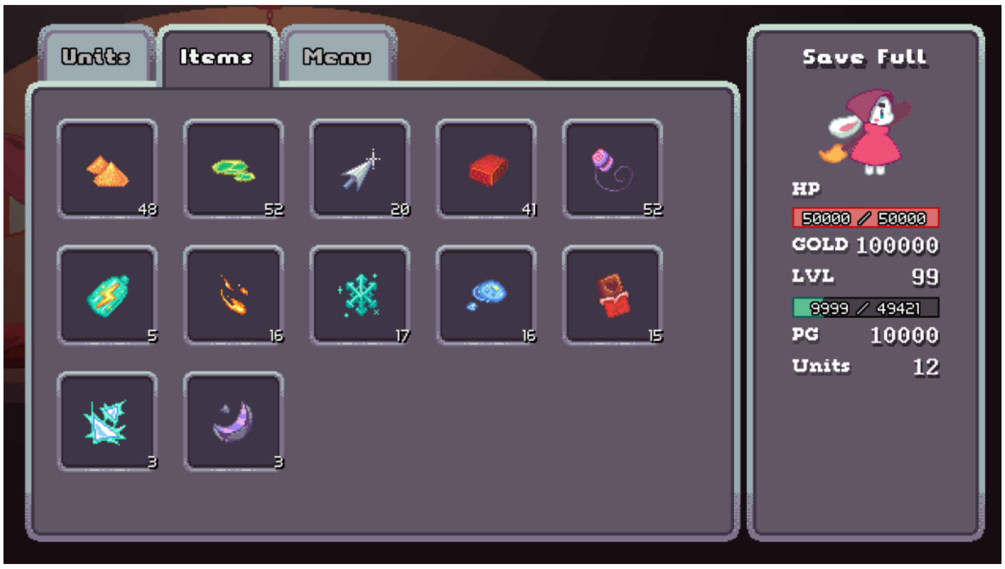
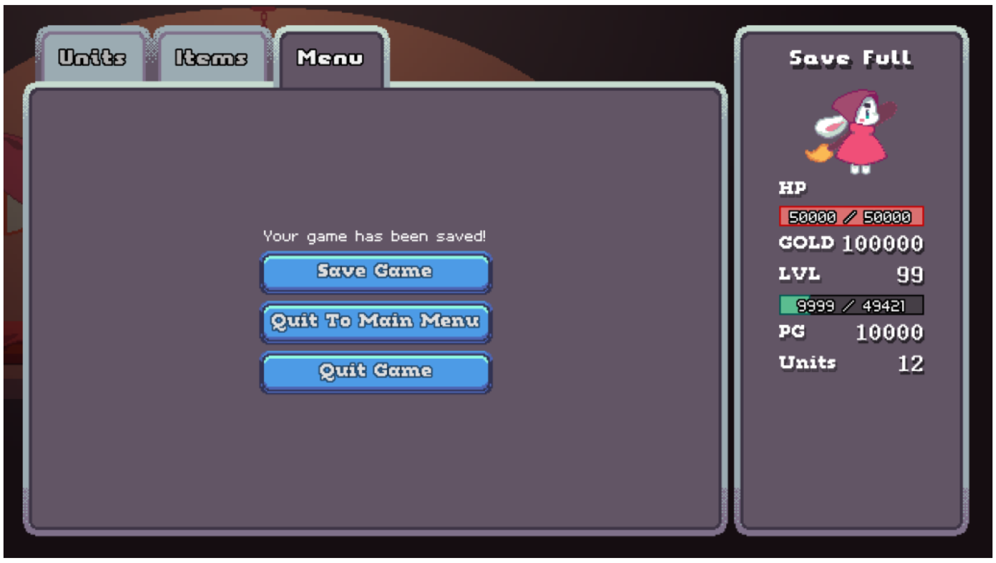


Figure 18: Items tab (Original work, 2024.)



Your game has been saved!

Figure 19: Menu tab (Original work, 2024.)

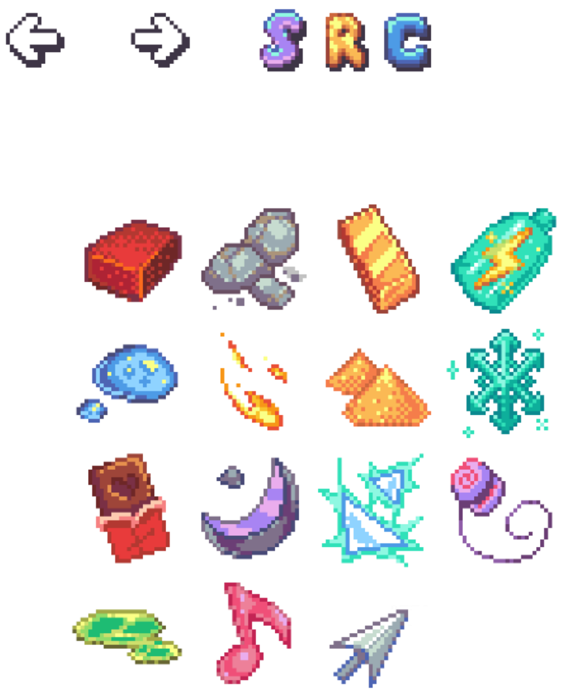


Figure 20: Finished material sprites and assorted icons (Original work, 2024.)

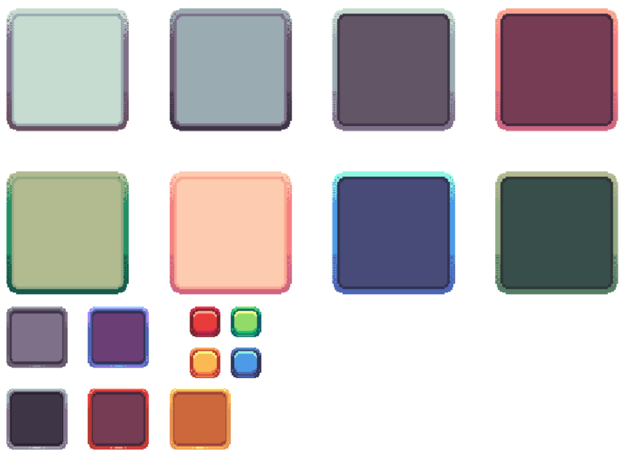


Figure 21: Sprites used for UI backgrounds, panels and buttons (Original work, 2024.)

Both icons and UI backgrounds are implemented using `AtlasTexture`s. An `AtlasTexture` is a resource type in Godot which allows the developer to use a small region of a larger image as the texture.

When interacting with the table, the player can access the gacha menu, shown in Figure 22. The player has the option of drawing 1 gacha at a time or drawing 5 gacha at a time.

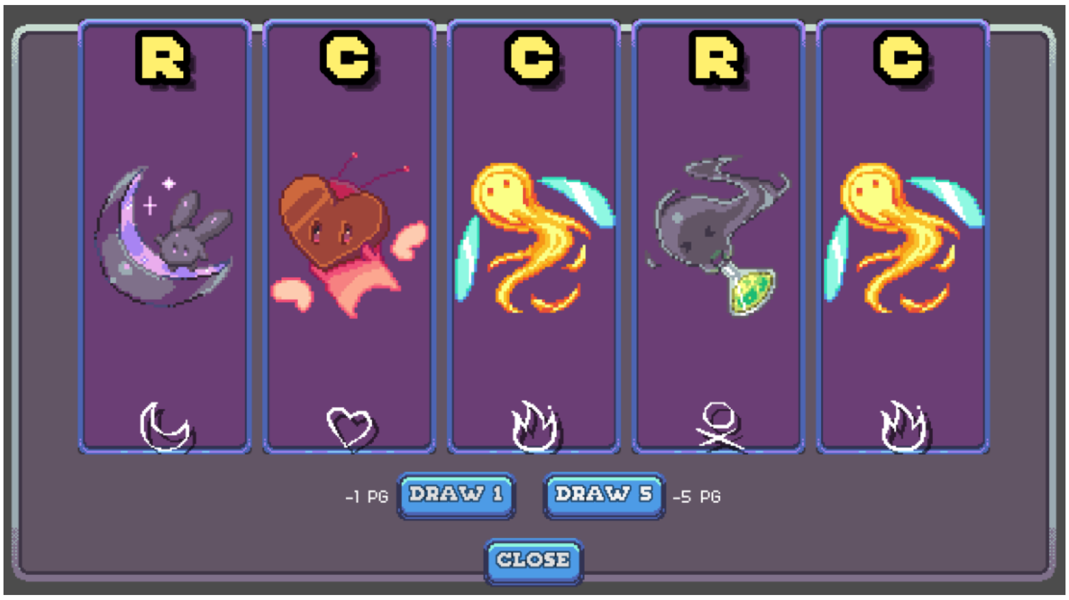


Figure 22: Gacha draw interface (Original work, 2024.)

4. Game Architecture

The following chapter is going to describe the game's architecture and underlying systems, with an emphasis on specific areas of interest. The chapter is split into 5 sub-chapters. The first chapter, *Ability System*, aims to explain the concept behind the ability system in the game. Following *Ability System* is the chapter *Scene Architecture* which shows and explains the scene tree setups of the areas in the game and the characters. After that comes the chapter *Singletons* which lists all of the main singletons in the game and their functions. Following *Singletons* is the chapter *Components*, which aims to explain the functionalities of several different custom component nodes in the game. Lastly, the chapter *Data Objects* explains the different custom resources in the game and how they are used.

4.1. Ability System

The ability system is one of the most modular and configurable systems of the game. It allows the developer to create new abilities by stacking together a series of effects, each with its own duration, stats, targets and chance to trigger. The effects are independent of each other, and the ability executes them in sequential order. This configuration allows abilities to range from simple one-shot attacks to multi-step executions with different targets. To illustrate, below is an example sequence of an ability in the game, which would make the target enemy take damage over time before scattering some damage to its neighbours:

1.) *Modulate Colour Effect* – it turns the targeted enemy blue to indicate the effect has started,
2.) *Change Stat Effect* – changes the `ATB_fill` stat of the targeted enemy's `ATB Component` to make it slower, to indicate that the enemy cannot execute actions due to being frozen,
3.) *Damage Over Time Effect* – applies 20 damage (`primary_stat`) every tick for the duration of 10 seconds (`secondary_stat`) to the targeted enemy,
4.) *Particle Effect* – plays particle effects over the targeted enemy to indicate that the *Damage Over Time* (shortened *DoT*) effect has ended,
5.) *Flash Effect* – similarly to above, makes the targeted enemy entity flash white to indicate that the enemy has broken out of its frozen status,
6.) *Modulate Colour Effect* – changes the enemy's colour back to normal,
7.) *Particle Effect* – uses `AOE_EXCLUDE_LIMIT` to target the enemy's neighbours and plays particle effects over them, indicating that the effect has spread,

8.) *Flash Effect* – uses `AOE_EXCLUDE_LIMIT` to make the target enemy's neighbours flash blue,

9.) *Direct Attack* – uses `AOE_EXCLUDE_LIMIT` to apply a burst of damage to the target enemy's neighbours,

10.) *Change Stat Effect* – changes the originally targeted enemy's `ATB_fill` back to normal,

11.) *Wait Effect* – waits 0.5 seconds until the effect can signal that it has finished, thus finishing the ability's execution.

Each ability has its own glyph or gesture that it gets triggered by. Abilities get instantiated as `AbilityNode` nodes and as the `BattleManager` singleton's children. This setup makes abilities execute independently of the targets, which also allows each ability to have different targets for each step of execution. A drawback of this approach is the fact that enemies lack awareness of which abilities they're currently being targeted by; there is no easy way to introduce limits on the number of currently applied status effects.

Removing effects works by iterating through all of the child nodes of `BattleManager`, and checking if each ability's effects share the same targets. If the effect shares the same targets as the targets on which the ability is to be removed, an `effect_finished` signal is emitted on those nodes, cutting the effect's duration short and essentially removing it from the target.

4.2. Scene Architecture

4.2.1. House Scene

`CanvasModulate` darkens the entire screen by a specified colour, giving the room a darker appearance. The `NonInteractible` node holds sprites for the non-interactible objects in the house, and the collision shapes for the floor and walls. The `Interactible` node holds all of the interactable objects within the scene. `WorldEnvironment` is a node that's necessary for the scene to have lighting, and allows the `PointLight2D` node to give off light.

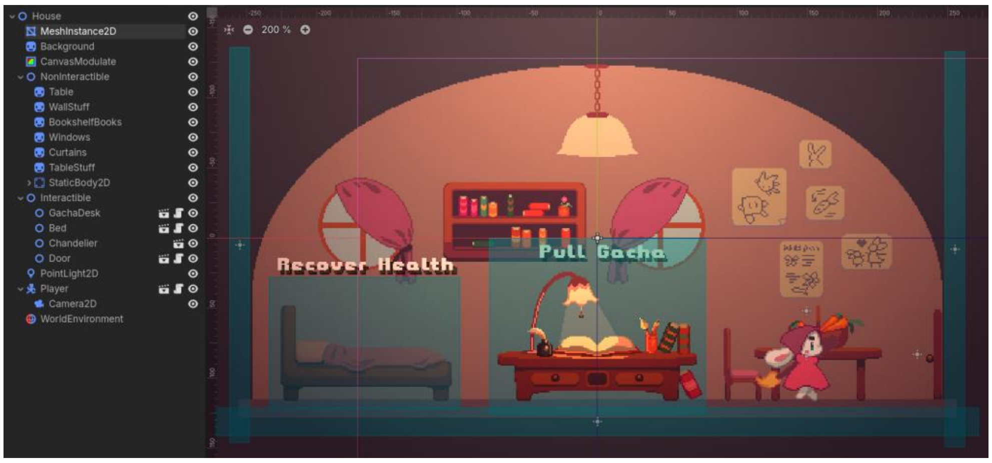


Figure 23: House scene tree (Original work, 2024.)

4.2.2.Interactable Objects

Each interactable object's root node is a node of type `InteractableItem`, or a node that inherits from `InteractableItem` and overrides its `_on_interaction` and `_on_interaction_exited` functions. `InteractableItem` requires a reference to an `InteractableArea`.

The `Label` shows up when the player walks into the interactable object's `InteractableArea`. The `InteractableArea` has an array of objects `toggled_when_near`. The objects in `toggled_when_near` are hidden by default. When the player enters the `InteractableArea`, it iterates through each object `in toggled_when_near` and makes the object visible to indicate an interaction is possible.

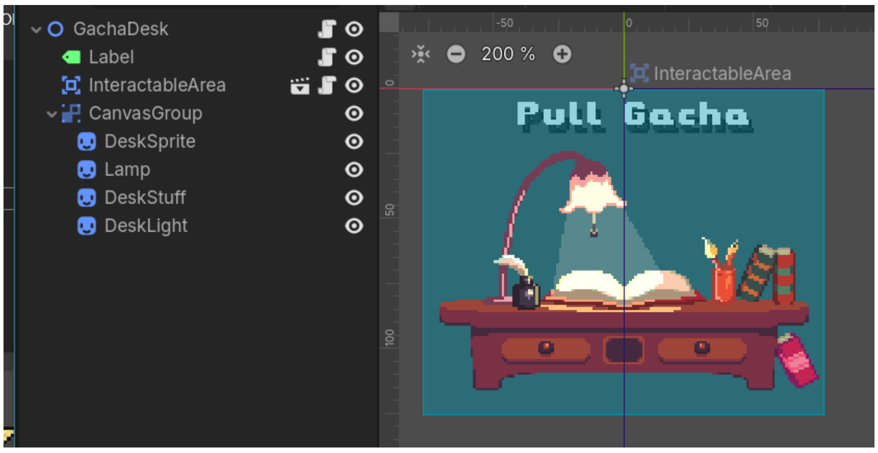


Figure 24: Interactable object scene tree (Original work, 2024.)

4.2.3.Overworld Scene

The overworld scene tree can be split into three sections: the background, the environment, and the actors.

The background is comprised of a single `TextureRect` for the background colour, and several different `Parallax2D` nodes for parallax layers.

The environment (represented by the `Environment Node2D` in the scene tree) consists of a `TileMapLayer` and `HomeTP`. The `TileMapLayer` represents the collision environment built using tiles. The `HomeTP` node is an `InteractableArea` that allows the player to teleport back to the house on interaction.

The actors represent the actors in the level. The player can freely move around the level with the camera following them. The enemies are static and cannot move. When the player comes into contact with the enemies, the game switches to the battle scene.

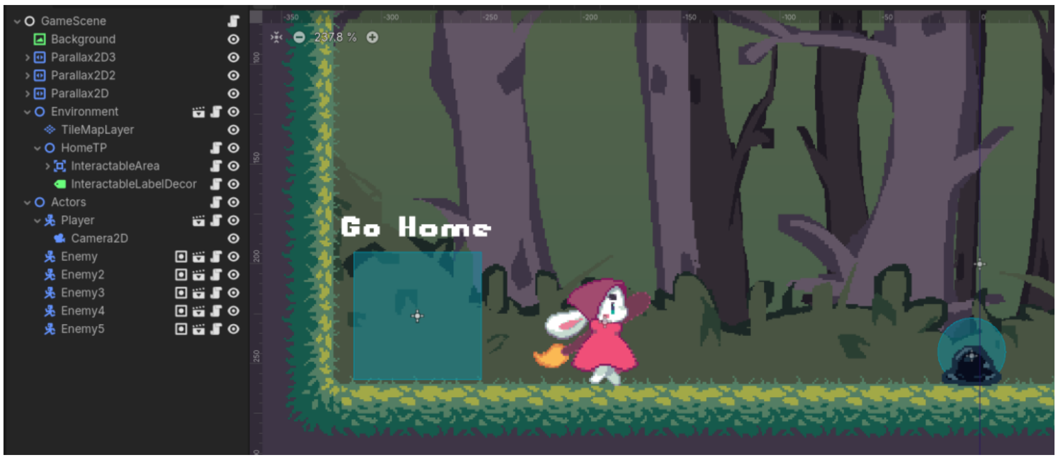


Figure 25: Overworld scene tree (Original work, 2024.)

4.2.4.Overworld Player

The player's root node is a `CharacterBody2D` node, which allows the developer to implement custom movement code. The root node requires references to a battle trigger `Area2D`, `AnimationPlayer`, `Sprite2D`, and `Health Component` to function properly. `OverworldBattleContact` is responsible for detecting whether the player has collided with an `OverworldEnemy`.

The user interface scenes are on the player entity, as this allows the player to easily open the menu from anywhere within the game outside of battle.

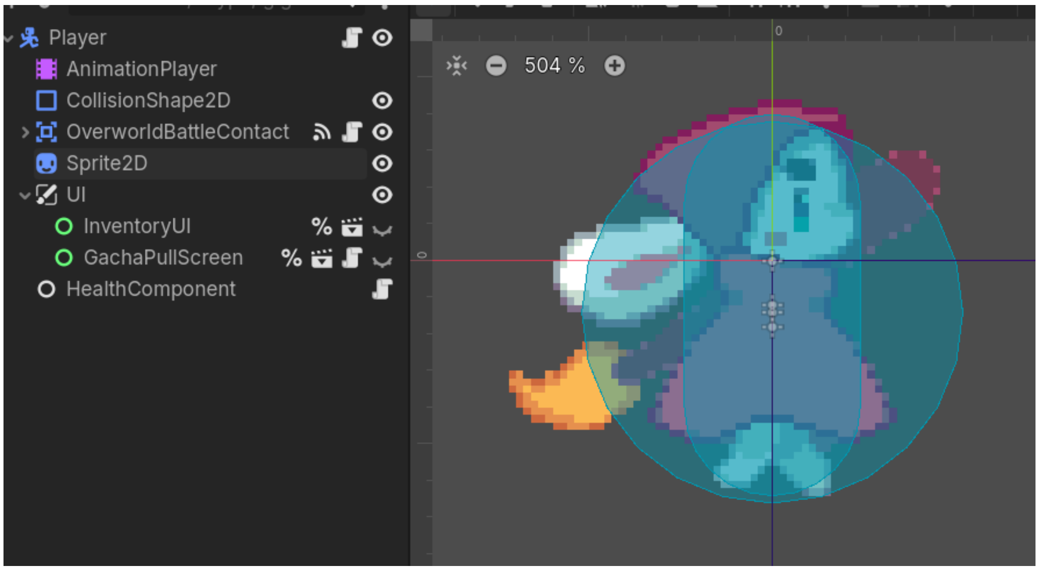


Figure 26: Overworld player's scene tree (Original work, 2024.)

4.2.5.Battle Scene

The `BattleScene`, similarly to the overworld scene, has a lot of parallax layers. The `BattlePlayer` and `BattleEntities` are found on the second to last parallax layer. The player is able to move the camera slightly with their mouse.

The `BattleEntities` node generates a random number at the start of battle and spawns the corresponding number of enemies in the scene.

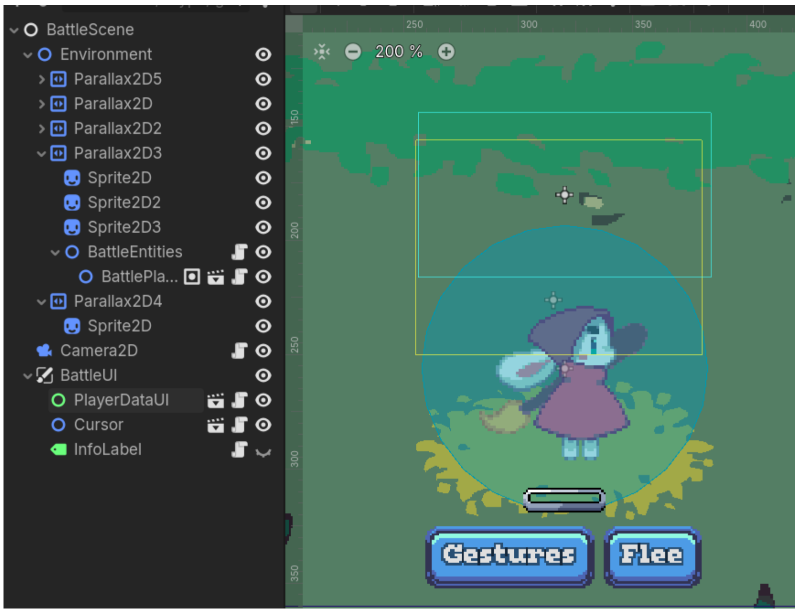


Figure 27: Battle scene tree (Original work, 2024.)

4.2.6.Battle Entity

The `EnemyBattleEntity` root node requires references to an `EnemyData`, an `AttackLabel`, an `ATB Component`, a `Sprite2D`, and a CollisionShape2D to function properly. The `AttackLabel` displays the attack the enemy has chosen to execute. The `CollisionShape2D` determines the area within which the player can draw gestures over enemies. The `Sprite2D` displays the enemy unit's art as a sprite. The `ATB Component` is responsible for ticking down the enemy's ATB gauge and sending signals to execute attacks.

When instantiated, the `EnemyBattleEntity` takes in the `EnemyData`'s information and initializes the appropriate components with it. All battle enemies use the same scene but can look and act differently according to the type of `EnemyData` the root node holds.

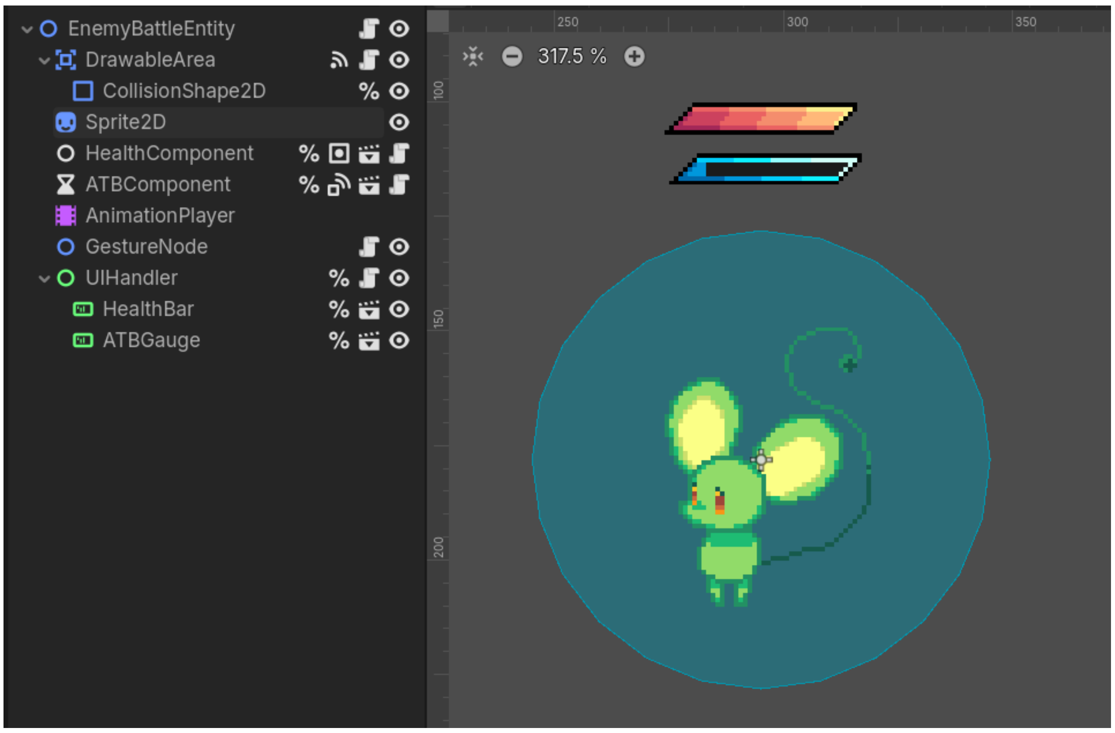


Figure 28: Battle Enemy's scene tree (Original work, 2024.)

4.3. Singletons

The following systems have been implemented using *Autoloads* [12], Godot's version of the singleton pattern [13]. Autoloads are scripts or scenes that have been set to be automatically instantiated at the top of the Scene Tree on runtime, and are able to be referenced and called upon within any script no matter its location without having to instantiate the classes first.

4.3.1. The Gesture Recognizer

The game's main battle mechanic relies on gesture recognition. Instead of traditional turn-based gameplay, players attack enemies using drawn gestures. The gesture recognizer within the game is based off the *$Q stroke-gesture recognizer* [14], modified to fit the game's needs.

The recognizer itself is implemented as a singleton called `QPointCloudRecognizer`, which has a function called `classify`. The `classify` function takes in a `Gesture` custom resource, compares the gesture to each of the pre-set gesture templates using the greedy cloud match algorithm, and returns the name of the recognized gesture. The rest gets handled by the `GestureNode`.

The $Q recognizer implementation in Godot used in the project has been developed specifically for this game project, but has been published and is available on GitHub as well [15].

4.3.2. Event Bus

The `GlobalEventsBus` singleton is an implementation of the *event bus pattern* [16] that often appears in Godot projects. An event bus in Godot usually has signals defined in a global singleton script. Nodes from anywhere within the project can emit and connect to functions in the global script without having to implement the signals themselves. This approach is useful in keeping nodes decoupled and unaware of each other while still allowing necessary data and events to pass through project-wide. The `GlobalEventsBus` defines the following signals:

- `Gesture_classified` – passes through `gesture` of type `StringName` and owner of type `Node`
- `Gacha_unit_acquired` – passes through `unit_data` of type `GachaUnitData`
- `Triggered_battle` – passes through `scene_where_triggered` of type `SceneTree` and `battle_scene_name` of type `String`
- `Battle_ended` – passes through `has_fleed` of type `bool`
- `Loot_scene_ended`
- `Ability_upgraded` – passes through `ability` of type `AbilityData`
- `Unit_leveled_up` – passes through `unit` of type `GachaUnitData`

4.3.3.Global Data

The `GlobalData` singleton is responsible for sending signals regarding player data and modifying the player data resource. It holds a reference to the currently selected player data resource (of type `PlayerData`) and is responsible for handling registering new *gacha* and battle rewards to the player. The function defines its own signal `player_data_changed` that gets emitted whenever the currently selected `PlayerData` resource gets modified.

Many different signals are subscribed to the `player_data_changed` signal in order to correctly display and update UI functions. Scripts use the singleton to read and to modify the player data resource as well.

4.3.4.Ability Database

The `GlobalAbilityDatabase` singleton makes use of the *Godot Resource Groups* addon. It holds a `Dictionary` [17] of abilities called `abilities_dict`. When first instantiated, it loads all of the available gesture templates from the `ability_data` folder and saves them into a resource group. The resource group then gets loaded into the abilities array. The singleton iterates through each ability (of type `AbiltiyData`) in the ability array and saves the ability into the `abilities_dict`, using the ability's `belonging_glyph` property as the key and the `AbilityData` resource itself as the value for each key-value pair in the dictionary.

The singleton's `abilities_dict` is used for matching the drawn gesture to the gesture's corresponding ability and for the passing of the belonging ability along to whichever functions need it.

4.3.5. Scene Manager

The `GlobalSceneManager` singleton is responsible for handling the loading and switching of game scenes. Scenes are stored in a `Dictionary` called `scenes`, with the `key` being the scene name and the `value` being the file path to the *.tscn* file. It is possible to access any potential scene by simply specifying a `String` with the desired scene name.

The singleton, aside from changing scenes, also holds a resource of type `OverworldPersistenceData`. This resource saves the last position of the player in the world as well as the enemy which triggered the scene switch. In the case where the player successfully defeats all of the enemies in a battle, the overworld enemy that instantiated the battle gets removed upon loading the overworld again; if the player has lost or fled the battle, the `GlobalSceneManager` disregards the `OverworldPersistenceData` and loads the player's home scene instead, resetting the overworld enemies.

4.3.6.Battle Manager

The BattleManager singleton is responsible for handling all of the gameplay actions during the battle portion of gameplay. It has two custom defined signals, ability_recognized_on_target which passes along the recognized ability (of type AbilityData) and the target on whom the ability was initially cast on (of type BattleEntity), and updated_enemies_array which informs nodes of whether the number of targets in the scene has changed. It also holds an array of BattleEntity called entities_array and an array of EnemyData called enemies_killed. It also has a reference to the BattlePlayer node in the scene simply called player.

Whenever a new battle screen is loaded in, the singleton registers all of the entities in the scene, saves them in an entities_array array of type BattleEntity, and subscribes to the appropriate signals.

It also implements the return_targets function, which takes in two arguments: a BattleEntity called sender and an TargetType enum value target_type. It finds the index of the sender in the entities_array, saves it under sender_i and then runs a check for what type of TargetType was passed through to return a targets array of nodes:

- SINGLE – returns only the sender BattleEntity,
- AOE – short for "Area of Effect", it returns all of the entities on the battle field, excluding the player,
- AOE_LIMIT – it returns the sender entity, and the entity to the left and right of it. Or, it returns an array of entities at sender_i, sender_i-1 and sender_i+1 positions,
- AOE_EXCLUDE – it returns all of the entities on the battle field, excluding the player and the sender,
- AOE_EXCLUDE_LIMIT – it returns an array of entities in entities_array at sender_i-1 and sender_i+1, without the sender,
- AOE_INCLUDE_PLAYER – same as AOE, except it includes the player,
- RANDOM_INCLUDE_PLAYER – chooses a random target within entities_array, where the player is also included in the random selection,
- RANDOM_EXCLUDE_PLAYER – chooses a random target within entities_array, where the player is excluded in the random selection,
- PLAYER – returns only the player.

EffectNodes request a list of targets for effects to be applied on using return_targets.

Whenever an enemy's ATB gauge gets filled, the enemy executes an attack using the execute_enemy_ability function. The BattleManager receives the signal from the enemy, looks at the list of abilities the enemy can execute and picks a random one to execute. When an enemy dies, the BattleManager handles enemy death logic: spawning the appropriate death effects, and adding the enemy's custom resource to the enemies_killed array. The resources in this array will later be used to determine which rewards the player should get at the end of a battle.

The BattleManager adds abilities to the scene using the spawn_ability function. The spawn_ability function takes in an ability of type AbilityData and a target of type BattleEntity as arguments. It instantiates a new node of type AbiliyNode and assigns it the appropriate ability data and primary target. It then adds it to the scene tree as a child of the BattleManager singleton.

Finally, the BattleManager is also responsible for detecting whether the battle has finished or not. There are two conditions for winning a battle: there being only one entity left in the entities_array, and that entity being of type BattlePlayer. When these conditions are satisfied, the BattleManager declares the battle won, generates a LootResource using data from the enemies_killed array, cleans up the scene of stray child nodes, and switches the scene.

4.3.7. Gacha Manager

The GlobalGachaManager singleton is responsible for all gacha-related activities in the game. It generates a list of gacha units to give to the player when requested. The gacha is able to be tuned by the developer being able to set rarity values for each "tier" of rarity. There are 3 tiers of unit rarity: Common (C), Rare (R), and Super Rare (SR). In the prototype, Rare has a 30% chance, and Super Rare has a 5% chance of being drawn. The GlobalGachaManager also takes into account the player's "pity count" value, where if the number of concurrent draws without getting an SR rarity unit exceeds the pity count number, the player forcibly receives a SR rarity unit in order to introduce fairness and reduce frustration.

When the singleton is instantiated, it grabs a list of all of the available units and sorts them into their respective arrays according to rarity: gacha_C, gacha_R, and gacha_S. When the player plays the gacha, a random decimal number from 0 to 1 is generated. The singleton first checks the pity count value, and if the player has gone enough times without a SR unit it appends a SR unit to the gacha unit array and resets the pity count value. Otherwise, the random number gets compared to the draw probabilities for each tier of rarity, and if it falls within the interval, it picks a random unit from the corresponding rarity's gacha array and appends it to the results array. At the end of the function, the gacha result is returned as an array of `GachaUnitData` and the gacha reward is registered to the current player's `PlayerData`.

4.4. Components

Components are nodes that give their parent nodes specific functionalities. Components are meant to be modular and reusable across a variety of parent nodes, and generally do not rely on knowledge of other components in the scene tree.

4.4.1. Gesture Node

`Gesture` nodes, when attached to a `BattleEntity`, allow the player to draw and recognize gestures over the targets using the mouse. When detection is triggered, the `Gesture` node takes in the currently drawn gesture and runs a normalization process so the drawn gesture can be saved as a type of `Gesture` custom resource. Then, it calls the `QPointCloudRecognizer` singleton and passes it the drawn gesture to get compared to the existing templates. When a match is found and returned, the `Gesture` node sends a signal containing its own parent (the target over which the gesture was drawn on) and finds the gesture's corresponding ability using `GlobalAbilityDatabase`. The returned `AbilityData` resource is then passed on to the `BattleManager` so it can get instantiated as an `AbilityNode`.

4.4.2. Health Component

The `HealthComponent` node is responsible for displaying and modifying the parent's custom resource during battle. The health component handles damage and healing calculations before changing the values on the parent resource. `HealthComponent` also has helper functions for changing the maximum possible health value, updating health UI displays, as well as spawning damage number labels. This allows the `HealthComponent` to be used on both the `BattlePlayer` and `EnemyEntity` instead of resources having to implement their own functions, as well as allowing `AbilityEffects` to simply interact with the `HealthComponent` of whichever entity it targets instead of having to run additional checks.

4.4.3. ATB Component

The ATB system in the game is implemented using `ATBComponent` custom nodes. `ATBComponent`s inherit from the `Timer` class, allowing them to run timers, pause and un-pause timers, and signals whenever a timer has run out. On initialization, the `ATBComponent` grabs its parent's resource and sets the `ATB_fill` variable to the resource's `default_ATB_fill_speed`, if it has one.

The `ATB_fill` variable of type `float` represents the amount of time it takes for the `current_ATB` variable to increase by 1. Given that ATB gauges have a constant maximum value of 100, variation in time needed to fill the gauge is achieved by changing the length of `ATB_fill`.

4.4.4. Ability Node

The Ability Node is instantiated as a child of the `BattleManager` singleton node. It is initialized with an `AbilityData` resource. The Ability node iterates through each `EffectData` in the `AbilityData` effects array and spawns an `AbilityEffect` node matching the `EffectData` type.

When all of the `AbilityEffect` nodes have been instantiated, the `Ability` node starts executing the effects in sequential order, using a state machine-like setup. The `Ability` node subscribes itself to the `AbilityEffect`'s node `effect_finished` signal and starts executing the first child node. When the `Ability` node receives the `effect_finished` signal, it increases its internal counter and begins executing the next child's ability. It will do so until it finds no next child to execute. When that happens, the `Ability` node will conclude that the ability has finished executing and will delete all of its child nodes before deleting itself.

4.4.5.Ability Effect Node

`Effect` nodes shouldn't be used by themselves. Instead, for each resource that inherits from `EffectData`, there is a corresponding node that inherits from `Effect` and implements functionality that uses the data from the `EffectData` resources to affect entities. The `Effect` node has an `upgrade_data` variable of type `AbilityData` and an `effect_data` variable of type `AbilityEffect`, as well as a `targets` array. The `execute_effect` function is supposed to be overridden by any classes that inherit from it. The `Effect` node has its `upgrade_data` and `effect_data` set by its parent `Ability` node before getting initialized. After initialization, the `Effect` node populates its targets array using the `GlobalBattleManager` singleton's function `return_targets`.

The following nodes inherit from `Effect`:

- `ChangeStatNode` – for each target, it changes the corresponding stat by either `primary_stat` or `secondary_stat`, depending on the type of stat to change,

- `ModulateColourNode` – tints each target in the specified colour,
- `DamageOverTimeNode` – creates a timer that is set to `TICK_DURATION` length. When it times out, it applies `primary_stat` damage to each target, and increases its internal counter by 1. When the counter equals `secondary_stat` (which corresponds to length of the effect), it declares the effect finished and stops applying damage,
- `DirectAttackNode` – simply changes the health of every target by the specified amount
- `EntityShakeEffectNode` – makes each target's sprite shake. The shake lasts for `shake_duration` amount of time, the possible sprite offset is determined by `shake_intensity`, and the speed at which the sprites interpolate between the values is determined by `shake_speed`.
- `FlashEffectNode` – tints the targets with the specified colour for only a brief amount of time before returning the colour back to normal,
- `ParticleEffectNode` – spawns the specified particle effects scene over the targets and plays it,
- `RemoveEffectNode` – ends the execution of the specified effect over the targets,
- `WaitEffectNode` – acts as a "buffer" between effects by waiting for the specified amount of time before sending the `effect_finished` signal.

4.5. Data Objects

The game heavily relies on custom data containers, called *custom resources* [19] in Godot, to store specific values, run checks, and to determine which objects to instantiate. Therefore, it would be appropriate to list all of the different prevalent data types in the game and explain what each one does first, as the architecture heavily relies on these different custom resources.

Each custom resource will have its properties listed along with the data type, followed by any functions the resource might have.

4.5.1. Gesture

The `Gesture` custom resource is used for storing data that represents the gestures the player can draw. Gestures are also the templates which drawn gestures get compared to. The `Gesture` custom resource has the following variables:

- `Gesture_name` – `StringName`, represents the name of the symbol of the gesture

- `Points` – Array of `Vector3`, where the `z` value holds the stroke index instead of a position coordinate,
- `Points_int` – Array of `Vector3i`, where the points are remapped from float to int
- `LUT` – `Dictionary`, short for *Look-Up Table.* It is used for optimization in the gesture recognition algorithm.

4.5.2. PlayerData

The `PlayerData` custom resource is used for storing any information relevant to the player, including the player's stats and all of the units and materials they currently have. The `PlayerData` declares the following variables:

- `Player_data_name` – `String`, used for displaying save data names
- `Experience_curve` – `Curve`, representing the 2D curve along which the player should get their experience rewarded compared to their current level.
- `Acquired_abilities` – Array of `AbiltiyData, the abilities the player is able to execute in battle,`
- `Acquired_gacha` – Array of `GachaUnitData, the list of gacha units the player currently has,`
- `Acquired_inventory` – Array of `InventorySlot`, where each InventorySlot has its own `material` of type `MaterialData` and `amount` of type `int,`
- `Max_health` – `int,` maximum possible health at the player's current level,
- `Current_health` – `int,` the player's current health value,
- `Level` – `int,` the player's current level
- `Current_experience` – `int,` the player's current experience,
- `Gold` – `int`, the player's current amount of gold, used for upgrading units,
- `Papers` – `int,` the player's current amount of currency used to play gacha,
- `Player_pity` – `int,` representing the number of *gacha* rewards the player got without receiving a unit of Super Rare rarity. This is implemented to introduce some fairness mechanics to the *gacha* system, where the player must receive a reward of Super Rare rarity after a number of "unsuccessful" pulls.
- `Current_party` – Array of `GachaUnitData`

4.5.3. EnemyData

The `EnemyData` custom resource is used to represent an enemy. It gets added to a `BattleEntity` node upon instantiation, and then the `BattleEntity` node modifies its parameters according to the data in `EnemyData`. The `EnemyData` custom resource has the following variables:

- `Enemy_name` – `String`, the enemy's name,
- `Death_particles` – `PackedScene`, the `GPUParticles2D` scene that should be instantiated on the enemy's defeat,
- `Physical_health` – `int`, the amount of maximum health the enemy has,
- `Average_gold` – `int`, the average amount of gold the enemy awards upon defeat,
- `Average_experience` - `int`
- `Spirit_energy` – `int`, the number of papers the enemy should award upon defeat,
- `Default_ATB_fill_speed` – `float`, the default speed at which the ATB gauge should fill. Because all ATB gauges have the same maximum number, the fill speed gets varied by the number of "ticks" it takes to fill the gauge. The tick length is determined by dividing `default_atb_fill_speed` with an `ATB_DIV` constant, found in the `ATB Component`,
- `ATB_variability` – `float`, the potential variability of the ATB fill speed of the enemy, to reduce scenarios where groups of enemies all attack at the same time,
- `Enemy_abilities` – Array of `EnemyAbilityData`, listing which attacks the enemy could potentially execute.

4.5.4. GachaUnitData

The `GachaUnitData` custom resource represents all of the information that makes up a gacha unit. It implements the following variables:

- `Unit_name` – `String`, the name of the gacha unit,
- `Rarity` – `enum` of type `RarityRank`, possible values `COMMON = 1`, `RARE = 2`, `SUPER_RARE = 3`, used to determine the unit's rarity in the gacha,
- `Unit_ability` – `AbilityData`, the ability the unit unlocks,
- `Unit_art` – `Texture`, the unit's sprite,
- `Level_up_mat` – `MaterialData`, the material the unit uses to level up,
- `Current_level` – `int`, the unit's current level. Starts from 1.

`GachaUnitData` has two additional functions:

- `Level_up` – this increases the unit's current level, and samples a value along the abilities upgrade curve to upgrade the stats by,

- `Get_upgrade_cost` – returns an `int`. Samples a value along the abilities cost curve to determine how much the next upgrade will cost in gold.

4.5.5. MaterialData

The `MaterialData` custom resource is used to hold information about upgrade materials. Originally, it was intended for each material to have a shop purchase price, but the material shop hasn’t been implemented.

- `Material_name` – `String`, the material’s name,
- `Material_sprite` – `Texture`, the material’s sprite.

4.5.6.InventorySlot

The `InventorySlot` custom resource is used to represent a single slot of inventory space in the `PlayerData`. Due to the inventory system’s simplicity, the inventory itself is just an array of type `InventorySlot` on the player. The `InventorySlot` custom resource defines the following variables:

- `Amount` – `int`, the number of resources in the inventory slot,
- `Material` – `MaterialData`, the material the slot holds.

As soon as the player runs out of a specific material, the material’s belonging `InventorySlot` gets removed from the `PlayerData` and the inventory UI.

4.5.7.LootResource

The `LootResource` custom resource is generated at the end of every battle that determines how much gold, experience and papers should be awarded to the player. It has the following variables:

- `Experience_curve` – `Curve`, the curve along which experience awarded should scale,
- `Gold` – `int`, the number of gold to award to the player,
- `Paper` – `int`, the number of papers to award to the player,
- `Experience` – `int`, the number of experience points to award to the player.

The `LootResource` also has the following functions:

- `Generate_resource` – calls the `calculate_experience`, `calculate_gold` and `calculate_paper` functions to assign values to `experience`, `gold` and `papers,`

- `Calculate_experience` – grabs the player's current level using the `GlobalData` singleton's `player_data` property of type `PlayerData`, then iterates through the number of enemies defeated. For each enemy defeated, it samples a point along the experience curve using the following formula:

```
exp = experience_curve.sample(float(level)/100) * 100
```

and adds the result of `exp` to `experience`. The more enemies the player has defeated, the more experience they get,

- `Calculate_gold` – takes in an array of `EnemyData`, then iterates through each `EnemyData` in the array and sums up the amount of gold each enemy drops to award to the player,
- Calculate_paper – takes in an array of `EnemyData`, then iterates through each `EnemyData` in the array and sums up the amount of paper each enemy drops to award to the player.

4.5.8. AbilityData

The `AbilityData` cusom resource is used to describe abilities within the game and hold all of the information an ability needs to function. It has the following variables:

- `Ability_name` – `String`, the name of the ability,
- `Ability_description` – `String`, the ability's description,
- `Belonging_glyph` – `String`, the name of the gesture that triggers the ability to execute,
- `Primary_stat` – `int`, representing the additional number of points to add to each `AbilityEffect`'s `primary_stat`, essentially serving as an upgrade stat,
- `Secondary_stat` – `float`, same as above, meant to be summed with each `AbilityEffect`'s `secondary_stat`,
- `Chance_to_trigger` – `float`, same as above, meant to be summed with each `AbilityEffect`'s `chance_to_trigger`,
- `Effects` – Array of `AbilityEffect`, representing the chain of execution of `AbilityEffects` that make up the ability.

The `AbilityData` resource has the following functions to make it easier to modify resources:

- `Upgrade_primary_stat` – increases the `primary_stat` by `value`
- `Upgrade_secondary_stat` – increases the `secondary_stat` by `value`

- `Upgrade_trigger_chance` – increases the `chance_to_trigger` stat by `value`

4.5.9. EffectData

`EffectData` is a custom resource meant to represent a single "step" of an ability. It can have different target types and different primary and secondary values. `EffectData` has the following variables:

- `Target_type` – `enum` of `TargetType`, with the following possible values: `SINGLE, AOE, AOE_LIMIT, AOE_EXCLUDE, AOE_EXCLUDE_LIMIT, AOE_INCLUDE_PLAYER, RANDOM_INCLUDE_PLAYER, RANDOM_EXCLUDE_PLAYER, PLAYER.` A description of what each `TargetType` does can be found in section 4.3.6 – BattleManager.
- `Primary_stat – int, meant as the effect's primary minimum possible stat`
- `Secondary_stat – float, meant as the effect's secondary minimum possible stat`
- `Chance_to_trigger – float, meant as the effect's minimum possible chance to trigger`

`EffectData` by itself is not meant to be used. Instead, different types of effects should inherit from the `EffectData` resource, allowing them to use the basic `EffectData` properties while implementing their own. The following resources inherit from `EffectData`:

- `ChangeStatEffect` – changes the chosen `stat_to_change` by `primary_stat`
- `DamageOverTimeEffect` – has a `TICK_DURATION` constant, `variability` and `particle_effect,`
- `DirectAttack` – implements no special properties, changes the `current_health` by `primary_stat.` If the value is negative, it adds health instead,
- `EntityShakeEffect` – implements `shake_duration`, `shake_intensity` and `shake_speed` variables to determine in which manner an entity's visuals should shake
- `FlashEffect` – implements `flash_colour` of type `Color` and `flash_duration` of type `float` variables
- `ModulateColourEffect` – implements `colour` of type `Color`, to determine what colour to tint the entity, and `modulation_speed` of type `float`, to determine how long it takes to change the entity's `modulate` property to the desired `colour`
- `ParticlesEffect` – particles of type `PackedScene,` serves as decorative particle effects that should trigger at some point within the ability

- `RemoveEffect` – implements `effect_to_remove` with possible values of `"DoT" = 0` and `"Stat Change" = 1`,
- `WaitEffect` – implements `wait_time` of type `float`, serves as a decorative effect with its purpose only being to add additional time to the ability's execution or wait for other effects in the ability to finish before proceeding.

This lends itself to a highly flexible and configurable ability system, where a new data container for a new type of effect can be added simply by making a new script and making it extend `EffectData`.

5. Conclusion

Creating this prototype has made me consciously aware of the sheer amount of work and thought that goes into the development behind each genre of game.

Role playing games and gacha games are genres that require an incredible amount of effort, time and material to develop. Not only do they require a great amount of visual and art resources and writing to truly make them shine, each genre has a plethora of underlying systems that need to be taken into account during development. RPGs need turn queues, enemies, data management, stat management, modifiers, equipment, mechanics, items, and a plethora of other things depending on how complex you want the system to be. Gacha games often need to feature an incredibly vast and varied cast of characters, with each update bringing more to the game. That is without taking into account the networking implications of gacha games. When combined, the underlying architectures can get very complex and very large very quick. My own project wasn't nearly as complex as a full-fledged RPG, but still turned out to be the most demanding thing I've worked on so far.

During the development of this prototype, I have come upon many limitations, that due to time and knowledge constraints, that have forced me to scrap earlier parts of my progress or change the plan entirely. I have researched game mechanisms, code architecture, and different programming patterns in hopes of developing an easily expandable and modular game. There were no tutorials or guides for Godot – or other game engines – on developing expandable games with complex systems such as this, only broad guidelines.

I have learned that combining two incredibly demanding genres to develop such as RPGs and gacha games together is an incredibly demanding undertaking due to the technical requirements both genres have. Despite this, I have chosen this as my theme because I found myself incredibly inspired by the games mentioned at the start of the work. I consider gacha systems to be an incredibly powerful way of adding replay value and content to any kind of game genre, if handled responsibly. Figuring out and making each part of the game by hand was a hard but rewarding experience. Developing this prototype has taught me a lot about setting up a framework for making expandable games. Hopefully, the experience gained from this will aid me in further game development projects.

6. References

[1] G. Engine, ‘First public release!’, Godot Engine. Accessed: Aug. 18, 2024. [Online]. Available: https://godotengine.org/article/first-public-release/

[2] ‘GDScript reference’, Godot Engine documentation. Accessed: Aug. 18, 2024. [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/tutorials/scripting/gdscript/gdscript_basics.html

[3] ‘Godot’s design philosophy’, Godot Engine documentation. Accessed: Aug. 18, 2024. [Online]. Available: https://docs.godotengine.org/en/stable/getting_started/introduction/getting_started/introduction/godot_design_philosophy.html

[4] ‘Overview of Godot’s key concepts’, Godot Engine documentation. Accessed: Aug. 18, 2024. [Online]. Available: https://docs.godotengine.org/en/stable/getting_started/introduction/getting_started/introduction/key_concepts_overview.html

[5] ‘Observer · Design Patterns Revisited · Game Programming Patterns’. Accessed: Aug. 18, 2024. [Online]. Available: https://gameprogrammingpatterns.com/observer.html

[6] J. Thomä, *derkork/godot-resource-groups*. (Aug. 16, 2024). GDScript. Accessed: Aug. 18, 2024. [Online]. Available: https://github.com/derkork/godot-resource-groups

[7] don-tnowe, *don-tnowe/godot-resources-as-sheets-plugin*. (Aug. 16, 2024). GDScript. Accessed: Aug. 18, 2024. [Online]. Available: https://github.com/don-tnowe/godot-resources-as-sheets-plugin

[8] V. Gerevini, *viniciusgerevini/godot-aseprite-wizard*. (Aug. 17, 2024). GDScript. Accessed: Aug. 18, 2024. [Online]. Available: https://github.com/viniciusgerevini/godot-aseprite-wizard

[9] D. Capello, ‘Aseprite’. Accessed: Sep. 07, 2024. [Online]. Available: https://www.aseprite.org/

[10] T. Dang, ‘The addictive design of mobile gacha games’. Accessed: Jan. 31, 2024. [Online]. Available: http://www.theseus.fi/handle/10024/805479

[11] ‘Xenoblade Chronicles^{TM} 2 - Nintendo Switch - Games - Nintendo’. Accessed: Aug. 20, 2024. [Online]. Available: https://www.nintendo.com/au/games/nintendo-switch/xenoblade-chronicles-2/?srsltid=AfmBOop3Bd7blUO_Km4IaT9SlR0JTiWh_9lBOr3XL2hYR_Yvqs2UA3eN

[12] ‘Active Time Battle (Concept)’, Giant Bomb. Accessed: Aug. 19, 2024. [Online]. Available: https://www.giantbomb.com/active-time-battle/3015-95/

[13] ‘Resurrect 64 Palette’. Accessed: Aug. 20, 2024. [Online]. Available: https://lospec.com/palette-list/resurrect-64

[14] ‘Singletons (Autoload)’, Godot Engine documentation. Accessed: Aug. 18, 2024. [Online]. Available: https://docs.godotengine.org/en/latest/tutorials/scripting/tutorials/scripting/singletons_autoload.html

[15] ‘Singleton · Design Patterns Revisited · Game Programming Patterns’. Accessed: Aug. 18, 2024. [Online]. Available: https://gameprogrammingpatterns.com/singleton.html

[16] R.-D. Vatavu, L. Anthony, and J. O. Wobbrock, ‘$Q: a super-quick, articulation-invariant stroke-gesture recognizer for low-resource devices’, in *Proceedings of the 20th International Conference on Human-Computer Interaction with Mobile Devices and Services*, Barcelona Spain: ACM, Sep. 2018, pp. 1–12. doi: 10.1145/3229434.3229465.

[17] angrychill, *angrychill/q-dollar-gesture-godot*. (Aug. 15, 2024). GDScript. Accessed: Aug. 22, 2024. [Online]. Available: https://github.com/angrychill/q-dollar-gesture-godot

[18] ‘The Events bus singleton · GDQuest’, GDQuest. Accessed: Aug. 18, 2024. [Online]. Available: //gdquest.com/tutorial/godot/design-patterns/event-bus-singleton/

[19]‘Dictionary’, Godot Engine documentation. Accessed: Aug. 18, 2024. [Online]. Available: https://docs.godotengine.org/en/stable/classes/classes/class_dictionary.html
[20]‘Resources’, Godot Engine documentation. Accessed: Aug. 19, 2024. [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/scripting/tutorials/scripting/resources.html

7. List of Figures

8. List of Tables