VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Game programming (CO3045)

Assignment Report

"Chronicles of Algorithmia" - Algorithm Quizzes RPG 2D Game

Advisor(s):	Phan Tran Minh Khue	
Student(s):	Nguyen Kieu Bao Khanh	2152654
	Vo Hoang Phuc	XXXXXX
	Pham Duc Minh	2152770

HO CHI MINH CITY, DECEMBER 2024

Acknowledgement

Something

Member list & Contributions

No.	Full name	Contribution
1	Nguyen Kieu Bao Khanh	33%
2	Vo Hoang Phuc	33%
3	Pham Duc Minh	33%

Contents

1 Game Concept

1.1 Introduction

1.1.1 Serious games

Serious games are designed to achieve goals beyond mere entertainment, offering players an engaging way to learn, train, or solve problems. Unlike traditional games, they focus on practical outcomes, such as improving knowledge, building skills, or encouraging behavioral change. These games are often used in fields like education, healthcare, corporate training, and social awareness campaigns. For example, DuoLingo, a widely popular language-learning app, uses game-like features to make studying new languages enjoyable and effective.

Figure 1.1: Doulingo App

Another well-known serious game, Foldit, allows players to solve puzzles related to protein folding, contributing directly to scientific research. By combining fun gameplay

mechanics with educational or societal goals, serious games create an environment where learning feels rewarding and natural.

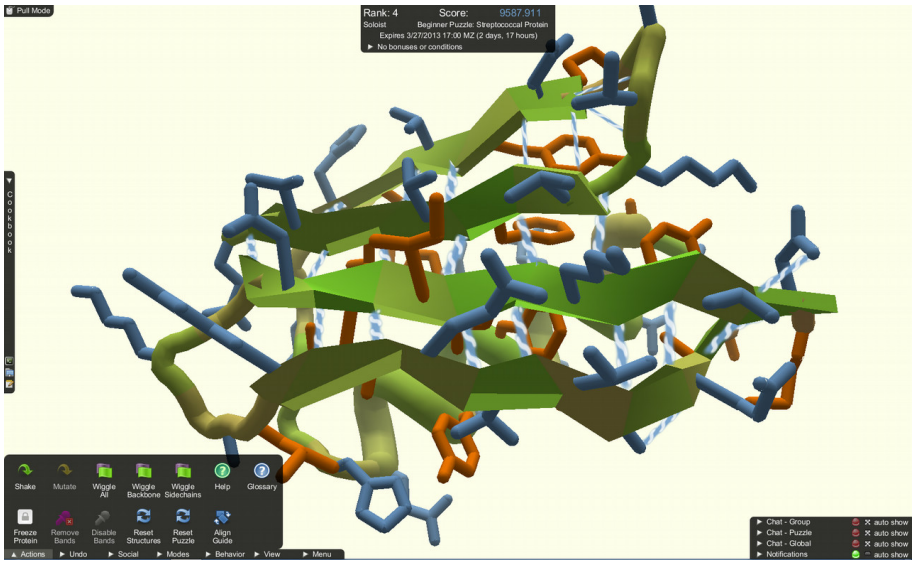

Figure 1.2: Foldit - Puzzle game about protein folding

One of the biggest strengths of serious games is their ability to immerse players in interactive and hands-on learning experiences. They allow users to experiment, fail, and try again without real-world consequences. For instance, business simulation games, such as SimVenture, help aspiring entrepreneurs develop their decision-making skills by running a virtual company. This method of learning helps users grasp concepts more deeply because they are actively participating rather than passively absorbing information.

Figure 1.3: SimVenture Game

In recent years, serious games have become more advanced and widely adopted due to technological innovations and the shift toward digital learning. Virtual reality (VR) has enabled the creation of immersive environments where players can engage in lifelike scenarios, such as firefighter training or military simulations. Artificial intelligence (AI) has also allowed these games to adapt to individual learning paces, offering personalized challenges and feedback. Platforms like Minecraft: Education Edition are now used in classrooms worldwide to teach subjects like history, math, and coding through creative building and problem-solving activities. Meanwhile, fitness-focused games like Ring Fit Adventure combine physical exercise with RPG-style challenges, promoting health and wellness in a fun way.

Figure 1.4: Minecraft: Education version

As industries recognize the effectiveness of serious games, their applications continue to expand. In healthcare, games like Re-Mission, designed for young cancer patients, encourage adherence to treatment by making players feel empowered and informed.

Figure 1.5: Re-Mission: first scientifically proven beneficial game for cancer patients

1.1.2 2D RPG Games

2D RPG (Role-Playing Games) are games where players explore worlds, follow stories, and take on the roles of heroes or adventurers. These games are known for their simple, flat graphics but focus on fun gameplay and strong stories. Players can solve puzzles, fight enemies, or complete quests to grow their characters and move the story forward. Many

people enjoy 2D RPGs because they are easy to play and don't need powerful devices to run, making them great for players of all ages.

A good example of a 2D RPG is Undertale, a game with a simple look but a very creative story. In the game, players can choose to fight monsters or befriend them, which changes how the story ends. The characters are funny and emotional, and the music adds to the experience. Even though it looks simple, Undertale shows how 2D RPGs can be very deep and memorable. This is why many people love 2D RPGs—they focus on making the game fun and meaningful.

Figure 1.6: Undertales game - Conversation with Sans

Our game, **Chronicles of Algorithmia**, follows this tradition while adding a new addition that blends education and entertainment. Set in the world of Algorithmia, players take on the role of the Debugger, a hero tasked with saving the land from the chaos caused by The Glitch. Along the way, players solve puzzles, battle enemies, and answer fun quizzes about data structures and algorithms. The game combines the charm of 2D RPGs with the chance to learn programming concepts in an interactive and exciting way, making it perfect for both gamers and learners alike.

1.1.3 Godot

The Godot Engine is a free and open-source tool for making video games. It was created in 2007 by two developers, Juan Linietsky and Ariel Manzur, to help their studio build games more easily. In 2014, they decided to release it for free under an open license so anyone could use or improve it. Over the years, Godot has grown into a popular game engine, supported by a community of developers who add new features and make it better.

Figure 1.7: Godot - Game Engine

Godot is packed with useful tools to help developers make games. It supports both 2D and 3D game development, with tools specifically made for each. The engine uses **GDScript**, a simple and easy-to-learn programming language, but it also supports other languages like Csharp and VisualScript. Godot has a user-friendly editor, a system for creating animations, and tools for testing and debugging games. It allows developers to make games for different platforms, including PCs, phones, and web browsers, without needing extra tools.

One big advantage of Godot is that it's completely free, so anyone can use it, even without a budget. It's lightweight, meaning it works well on older computers. Its 2D tools are some of the best available, making it perfect for creating 2D games. However, Godot has a few downsides. Its 3D tools are not as advanced as those in other engines like Unity or Unreal, which might limit developers making 3D games. Also, because it has a smaller community, finding certain add-ons or answers to questions can sometimes be harder.

Godot is very good at making 2D games. It treats 2D development as an important part of the engine, not just an extra feature. It has a special 2D editor where you can easily create and arrange scenes. Godot includes tools like tilemaps for creating levels, parallax

scrolling for depth effects, and an animation editor for adding movement to characters or objects. You can also use shaders to create cool visual effects, like lighting or glowing objects. Whether you're making a simple puzzle game or a detailed adventure, Godot makes 2D game development easy and fun.

1.1.4 Project scope and Purposes

- **Purpose of the Project**: The goal of this project is to create a fun 2D RPG game that helps players learn the basics of data structures and algorithms. The game is designed for beginners or students who might find these topics difficult to understand in a traditional classroom setting. By using puzzles, battles, and interactive challenges, the game makes learning easier and more exciting. Players will explore concepts like binary trees, sorting methods, and recursion while solving problems in an enjoyable way.
- **Scope of the Project**: The scope of this project includes the development of a functional game prototype that blends educational content with gameplay. Key elements of the project include:
  - Gameplay Mechanics: A mix of battles, puzzles, and NPC interactions where players must answer algorithm and data structure-related questions to progress.
  - Interactive Quizzes: Embedded quizzes tailored to reinforce learning, with feedback and explanations to help players understand their mistakes.
  - Level Progression: A structured level system that gradually introduces more complex concepts, ensuring a smooth learning curve.
  - Story Integration: A compelling narrative that immerses players in the world of Algorithmia, connecting educational challenges to the storyline.

1.2 Concept Overview

1.2.1 Story Concept and Narrative

The title, Chronicles of **Algorithmia**, reflects the combination of storytelling and learning about algorithms. "Chronicles" suggests a journey full of challenges and growth, while "**Algorithmia**" represents a world built around computer science concepts like data structures and algorithms.

In **Algorithmia**, players take on the role of the **Debugger**, a hero chosen to restore balance to the world after a mysterious force called **The Glitch** starts disrupting the land.

As the Debugger, players explore different regions, each focused on a specific computer science concept, such as binary trees or sorting algorithms. Along the way, players meet quirky NPCs who will quiz them on these topics and offer help or rewards based on the answers. These NPCs are there to guide players and make learning fun by incorporating questions and challenges into the story.

As the Debugger journeys through the world, they must face powerful **mini-bosses** that represent different algorithmic challenges. For example, one mini-boss could be a corrupted binary search tree, where players need to correctly sort data to defeat it. These mini-bosses test the player's knowledge of the algorithms they've learned in a way that feels like part of the adventure.

Throughout the game, NPCs will **quiz players on various concepts** like binary trees, sorting methods, and recursion. These quizzes are not only a fun way to learn but also help players progress in the game, rewarding them with experience points or special items when they answer correctly.

There are also side quests in the game that provide extra learning opportunities. These quests are optional but offer rewards and help players get better at using algorithms. For example, one side quest might involve finding pieces of a broken algorithm and solving problems along the way, like finding the shortest path through a maze. Another quest could involve using algorithms like divide and conquer or greedy algorithms to solve complex puzzles. These side quests give players a chance to explore more of the world, meet new characters, and dive deeper into algorithmic concepts.

1.2.2 Target players and platforms

Target Players:

- Aspiring programmers and computer science students seeking an engaging way to learn foundational concepts.
- Gamers who enjoy platform RPGs with a unique educational twist.
- Educators looking for innovative teaching tools.

Target Platforms:

- Primary: PC (Windows, macOS, Linux)
- Future Expansion: Mobile devices for quizzes and light gameplay adaptations.

2 Game Design

2.1 Game Mechanics

2.1.1 2D RPG Mechanics - Actions and Quizzes

- Player mechanism: movement, item interaction, enemies hitting, inventory management.
- Enemies behavior: movement, interactions with player, item drop.
- Map generation: procedural level generation, placing objects, obstacles.
- Item interaction: collecting, using, and managing items like keys or potions.
- Game progression: level transition, save and load game progress, win/lose condition.
- User interface: display game status and game menu for gamers.

2.1.2 Progressing Mechanics

The game's level progression gradually increases in difficulty. Early levels introduce basic mechanics, such as movement and item collection, while later levels challenge players with more difficult puzzles and enemies (Goblins instead of Slimes, for instance) that require a higher level of reasoning and combat. The Chronicles of Algorithmia (CoA) currently has 2 levels: Dungeon and Grass Fields. Each has different levels of difficulty, and the Dungeon has a boss fight when the player reaches the end of the Dungeon. Levels feature hidden areas with bonus items or optional quizzes to encourage exploration. These areas reward curious players with extra resources or insights into the game's story.

2.2 Level Design

2.2.1 Region themes

The game features multiple regions, each themed around specific computer science concepts.

For example, **Nodewood Plains** – A play on nodes in trees, representing the foundation of data structures in a peaceful, grassy plain with docile slimes and friendly NPCs (occasionally Goblins for fighting, but mostly slimes).

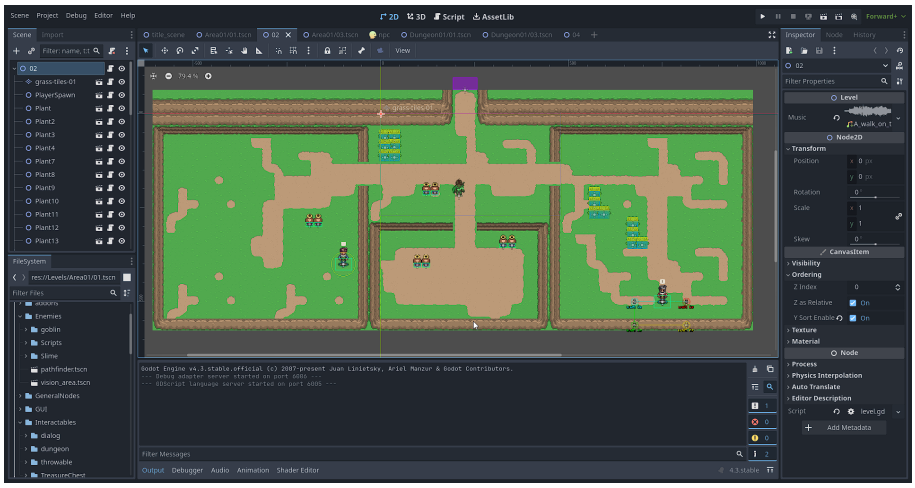

Figure 2.1: Nodewood Plains under development - Starting journey

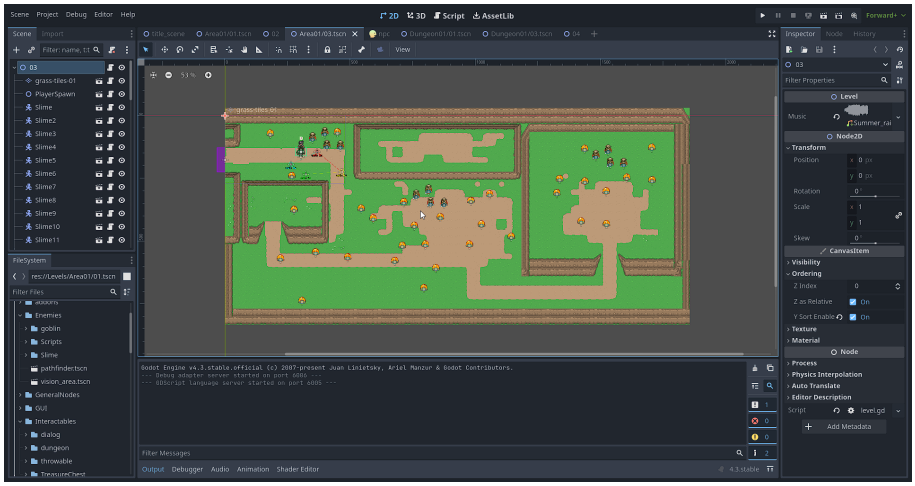

Figure 2.2: Nodewood Plains under development - Further into the plains

The Corruption Depth is a dark and eerie dungeon where The Glitch has taken over. The paths are broken and strange, and danger lurks around every corner. Slimes rampage and sneaky goblins try to stop you with their swords. Along the way, trapped NPCs will ask you quick questions about arrays and loops to test your skills and help them escape **The Corrupted Depth**.

Deeper inside, the challenges get harder, with tougher enemies and tricky puzzles blocking your way. At the very end, you'll face the **Dark Wizard**, a dungeon boss protecting an important piece of the code needed to save Algorithmia.

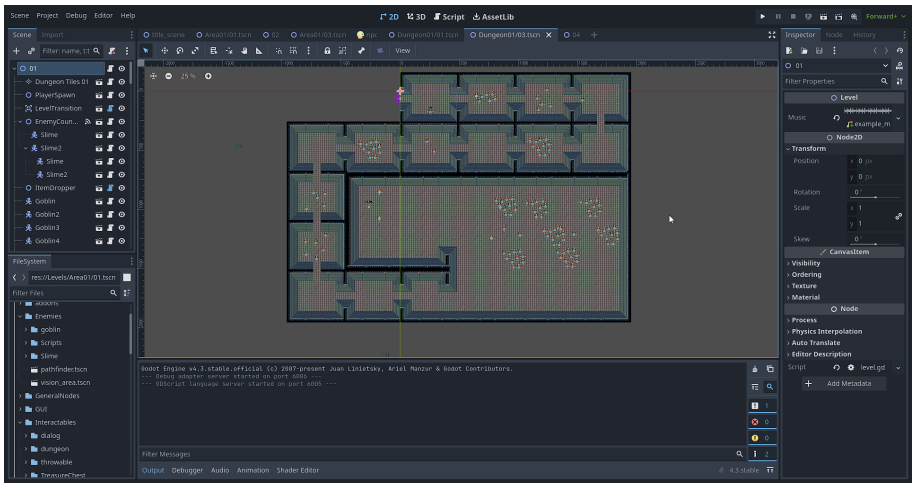

Figure 2.3: The Corruption Depth under development

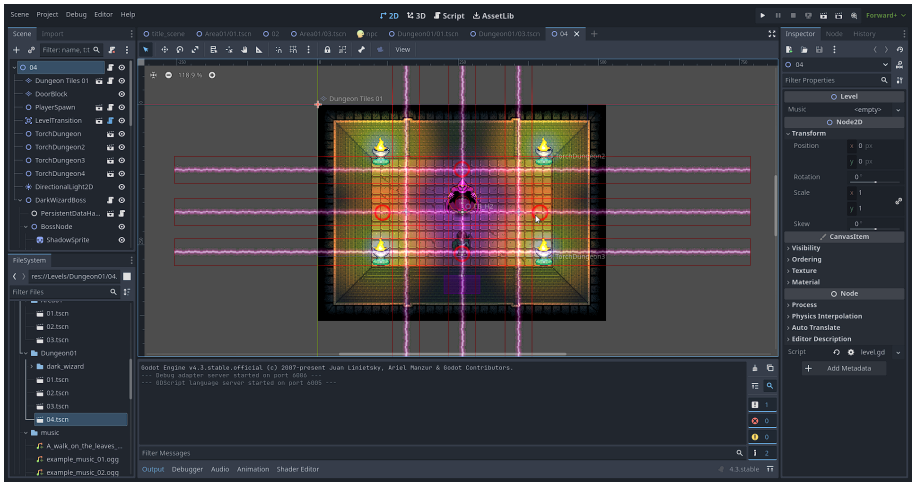

Figure 2.4: Boss - Dark Wizard Fight

2.2.2 Enemies

Enemies in the game range from basic minions to mini-bosses, each designed to align with the game's educational themes. Basic enemies, like slimes and goblins, introduce simple challenges early on, while mini-bosses and bosses, like the Dark Wizard, require players to have the basics of combat skills to fight them.

Figure 2.5: Enemies on the Nodewood Plains

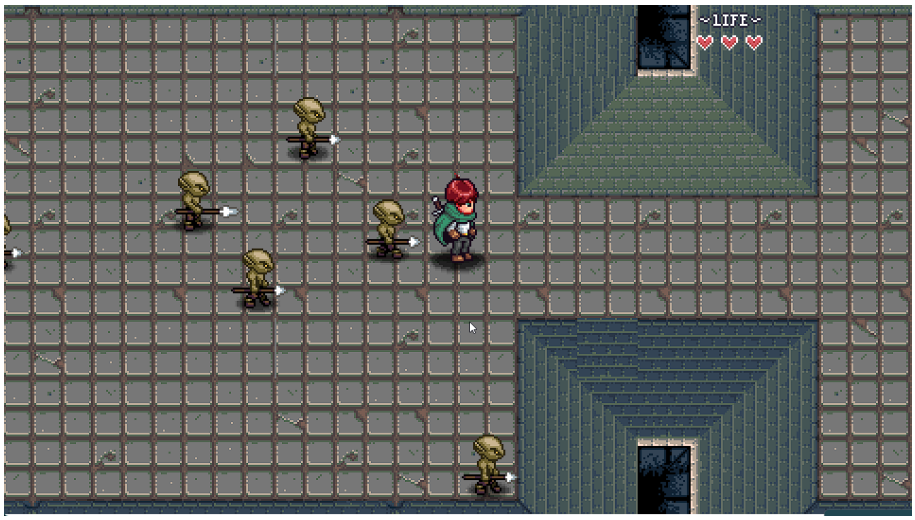

Figure 2.6: Enemies in the Corruption Depth

Enemies drop useful items upon defeat, such as apples, keys, or rare collectibles. This mechanic encourages players to engage with enemies strategically, balancing risk and reward.

Figure 2.7: Collectibles - Slime Essence

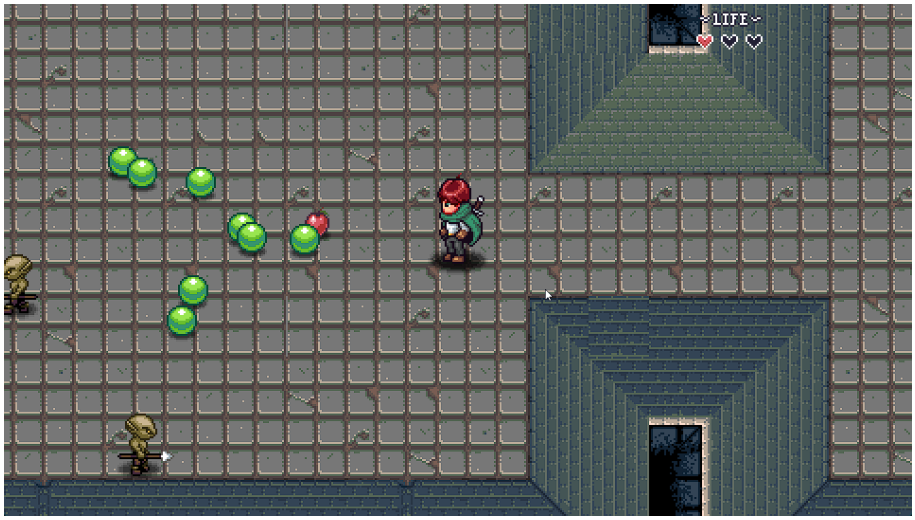

Figure 2.8: Collectibles - Apples

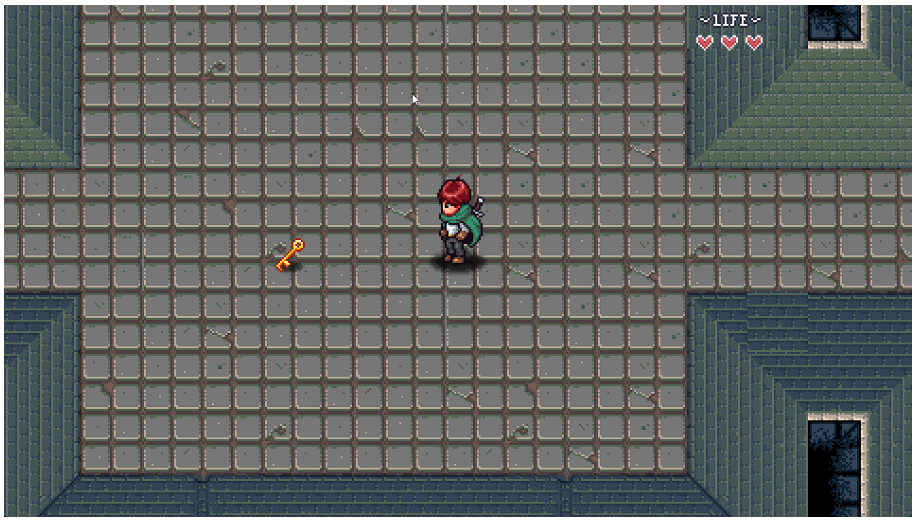

Figure 2.9: Collectibles - Key

The game also includes non-combat encounters with enemies, where players must solve puzzles or quizzes to disable them. These encounters provide variety, ensuring that the game's educational focus remains prominent even during combat-heavy sections. This feature is under development.

Figure 2.10: A NPC hints to the player that the weakness of some enemies are puzzles

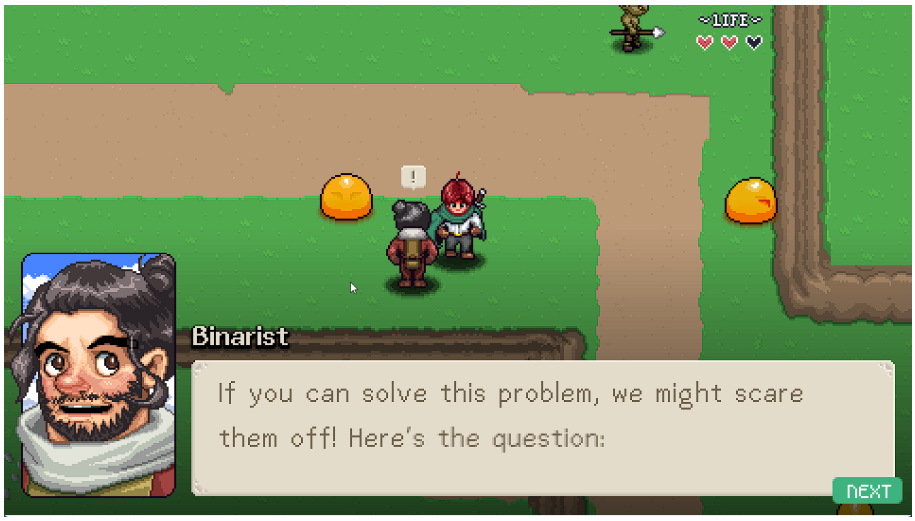

Figure 2.11: A NPC hints to the player that the weakness of some enemies are puzzles - 2

2.3 Graphical Design

2.3.1 Character Design - Hero

The game's visuals have a bright, fantasy look with colorful and fun designs. The hero, the **Debugger**, is a brave and simple character that players guide through the game.

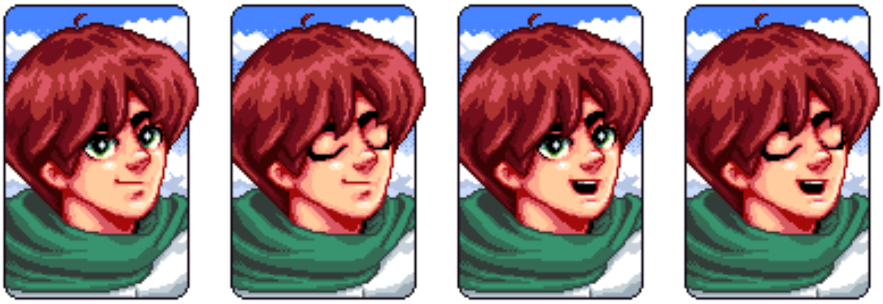

Figure 2.12: Player Portrait

Animations are smooth and clear, so actions like fighting enemies or picking up items feel good and easy to follow. Player Sprites includes 77 different frames for animation.

Figure 2.13: Player Sprites

- Attack Down: Frame 21 -> 23
- Attack Side: Frame 27 -> 29
- Attack Up: Frame 24 -> 26
- Carry Down: Frame 35
- Carry Side: Frame 41
- Carry Up: Frame 38
- Carry and Walk Down: Frame 42 -> 45
- Carry and Walk Side: Frame 50 -> 54
- Carry and Walk Up: Frame 46 -> 49
- Death: Frame 74 -> 76
- Idle: Down (0 -> 2), Side (6 -> 8), Up (3 -> 5).
- Walk: Down (9 -> 12), Side (17 -> 20), Up (13 -> 16)

2.3.2 Character Design - NPCs

NPCs stand out, like a wizard who explains algorithms using magical scrolls or a blacksmith who talks about tools in sorting. These creative designs make learning fun and exciting.

The Guide NPC is designed to feel helpful and wise, with a calm appearance that makes players trust them. He wears simple villager clothes. His kind expression and steady tone make them approachable, like a mentor who is always there to guide players.

Figure 2.14: The Guide Portrait

Figure 2.15: The Guide Sprites

- Idle Down: Frame 0
- Idle Side: Frame 2
- Idle Up: Frame 1
- Walk Down: Frame 3 - 4 - 5 - 6 - 5 - 4
- Walk Side: Frame 11 - 12 - 13 - 14 - 13 - 12
- Walk Up: Frame 7 - 8 - 9 - 10 - 9 - 8

The Binarist, on the other hand, is quirky and curious, reflecting their love for binary trees. He has the same assets but different colors than **The Guide**. His energetic personality makes learning from them fun, as he explains concepts with excitement and enthusiasm.

Figure 2.16: The Binarist Portrait

Figure 2.17: The Binarist Sprites

- Idle Down: Frame 0
- Idle Side: Frame 2
- Idle Up: Frame 1
- Walk Down: Frame 3 - 4 - 5 - 6 - 5 - 4
- Walk Side: Frame 11 - 12 - 13 - 14 - 13 - 12
- Walk Up: Frame 7 - 8 - 9 - 10 - 9 - 8

2.3.3 Character Design - Enemies

- **Slimes**
  - The slime sprites are squishy, colorful blobs with simple, bouncy movements. Their playful design hides their mischievous nature, as they split into smaller slimes when attacked, keeping players on their toes.
  - Sprites

Figure 2.18: Slime Sprite

- **Goblins**
  - The goblin sprites are small, sneaky creatures with sharp features and continuous animations. Their outfits and crude weapons show they've been corrupted by The Glitch, making them a crafty enemy.
  - Sprites

Figure 2.19: Goblin Sprite

2.3.4 UI Elements

- **Dialog System:** The Dialog System is designed to make conversations with NPCs engaging and clear. It uses a simple text box that displays messages, with smooth animations for transitions. Players can easily follow the story, answer quiz questions, and receive hints through this system. Special icons or colors highlight key information, like important terms or choices. We use BBCode to write the Dialogs and choices.

```
[wave]I am [b][color=blue]Algo[/color][/b],
your guide on this journey.[/wave]
```

Figure 2.20: Dialog between the Guide and The Debugger

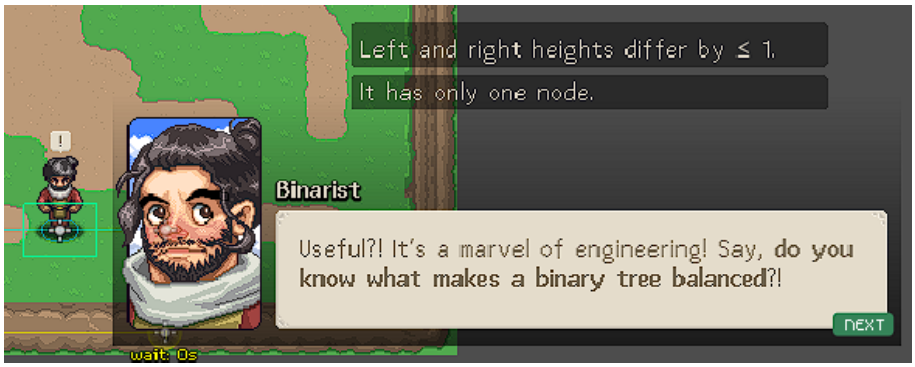

Figure 2.21: Dialog choices for quizzes

- Pause Menu, Inventories and Player Head-Up Display (HUD)

Figure 2.22: Pause Menu

2.3.5 Audio Resources

We use both free audio resources and composed music by our team members for each level. We composed our music with Musescore 3. MuseScore 3 is a free and powerful music notation software that helps users create, edit, and share sheet music. It offers an easy-to-use interface with drag-and-drop tools for adding notes, symbols, and dynamics.

Figure 2.23: Musescore 3 Banner

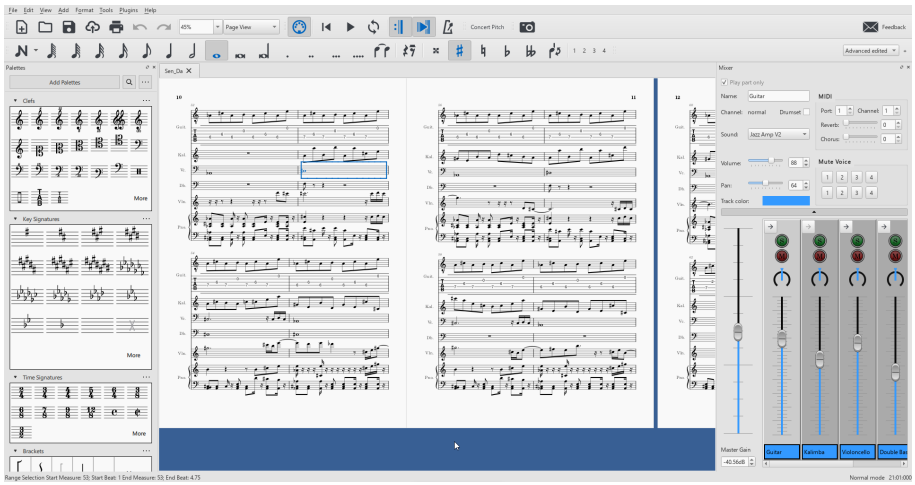

Figure 2.24: Compose CoA's music with Musescore 3

3 Game Development

3.1 Architecture

This overviews a game architecture system using Godot's game engine. Three nodes in progress: game script, material assets, and game scene. When the user interacts with the game scene, it will send information directly to the game engine, then trigger the game script to process the game state. In Godot, the game script is written using GDscript, which is a descriptive programming language used to process or access game materials. The final step of this process will send all these assets to the game scene and allow users to start new actions with the game.

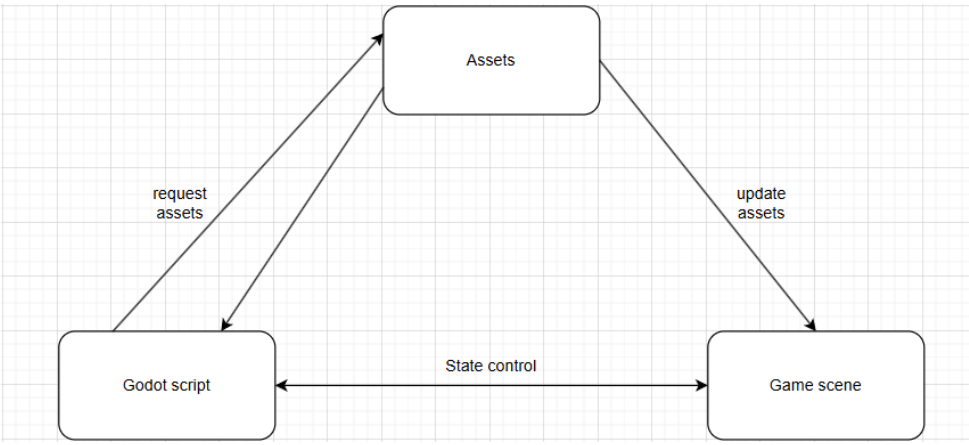

Figure 3.1: Caption

3.2 Design Patterns

3.2.1 Finite State Machine

The State pattern is a behavioral design approach that enables an object to modify its behavior dynamically based on changes to its internal state. It can appear as though the object's class itself has changed.

State Pattern: Problem Statement The State pattern is deeply connected to the concept of a Finite-State Machine (FSM):

Finite-State Machine: A program can exist in a limited number of predefined states at any given time. Each state dictates specific behavior, and the program can transition between these states based on predefined rules (transitions). However, not all state transitions are allowed; they depend on the current state. For example, consider a Document

class that can be in one of three states: Draft, Moderation, or Published. The publish method behaves differently for each state:

In Draft, the document moves to the Moderation state. In Moderation, the document is published if the user is an administrator. In Published, the method performs no action.

Managing states and transitions with conditionals (if or switch) can quickly become unwieldy as the number of states and state-dependent behaviors increases. Over time, this results in bloated, difficult-to-maintain code with scattered transition logic across methods.

Solution: The State pattern proposes isolating state-specific behaviors into separate classes and delegating responsibilities to these classes.

The primary object, referred to as the context, maintains a reference to a state object that defines its current state.

Instead of handling state-specific behavior, the context delegates this responsibility to the relevant state object.

Transitioning to a new state involves swapping the current state object with another that represents the desired state. This is possible as long as all state classes implement a shared interface, allowing the context to interact with them seamlessly.

While the State pattern resembles the Strategy pattern, the key distinction lies in state objects being aware of and able to trigger transitions between themselves, a behavior absent in Strategy.

Real-World Analogy: The behavior of a smartphone's buttons and switches changes based on its state:

When unlocked, pressing buttons performs various functions.

When locked, any button press leads to the unlock screen.

When the battery is critically low, pressing a button displays the charging screen.

When to Use the State Pattern:

When an object's behavior varies significantly based on its current state, especially when the number of states is large and changes occur frequently.

When a class becomes cluttered with numerous conditionals tied to state-specific logic.

When duplicate code exists across states and transitions, leading to poor maintainability.

By encapsulating state-specific logic within separate classes, the State pattern reduces maintenance overhead, simplifies code, and enables the introduction of new states without modifying existing code.

How to Implement the State Pattern

Define the Context Class: This class represents the primary object whose behavior depends on its state. Create a State Interface

Define methods specific to state-related behaviors.

Develop State Classes: Implement the state interface and extract state-specific logic from the context class. Address dependencies on the context's private members through public methods, nested classes, or other workarounds.

Update the Context Class: Add a reference field for the current state and methods to set or update the state. Replace conditional logic with calls to state methods. Switch States

Instantiate new state objects and assign them to the context's reference field.

Advantages and Disadvantages

Pros: Single Responsibility Principle: Keeps state-related logic modular by isolating it into separate classes. Open/Closed Principle: Allows new states to be added without altering existing code. Simplifies the context class by removing bulky conditionals. Cons Can be overkill for systems with only a few states or infrequent state changes. Relationship with Other Patterns The State pattern shares structural similarities with Bridge, Strategy, and Adapter. While all rely on composition and delegation, they address different problems:

The State pattern extends Strategy by enabling dependencies and transitions between states. Strategies, on the other hand, are independent and unaware of each other.

3.2.2 Singleton

The Singleton is a creational design pattern that ensures a class has only one instance while providing a unified access point to it.

Singleton Pattern: Problem Statement The Singleton pattern addresses two key issues simultaneously, albeit at the cost of violating the Single Responsibility Principle:

Ensuring a Single Instance of a Class: Sometimes, it's crucial to restrict the creation of multiple instances of a class, particularly when managing shared resources such as databases or files. For example, if an object is created once and then another attempt is made to instantiate it, instead of returning a new instance, the existing one will be reused. This behavior is not achievable with a standard constructor, as a new object is always returned by design. Providing Global Access to the Instance

Clients may unknowingly interact with the same instance repeatedly. While global variables can offer easy access to critical objects, they are risky since any part of the code can overwrite them, potentially causing application failure.

The Singleton pattern, akin to a global variable, enables access to a specific object from anywhere in the program. However, it safeguards the instance from being modified or replaced by other parts of the code. Moreover, centralizing the logic for creating a single instance within one class prevents this responsibility from being scattered across the application. This is especially beneficial if other parts of the code already rely on the Singleton.

In modern usage, the term "Singleton" is sometimes used to describe any implementation that addresses one of these problems, even if it doesn't address both.

Solution: To implement the Singleton pattern, the following two steps are essential:

Private Constructor: The default constructor is made private to prevent other objects from instantiating the Singleton class using the new operator. Static Creation Method

A static method is created to serve as the class's constructor. This method invokes the private constructor to generate an instance, stores it in a static variable, and returns the same cached instance for every subsequent call. This ensures that as long as the Singleton class is accessible, its static method can always return the same instance.

3.3 Implementation

3.3.1 Global Managers

1. Audio Manager (`global_audio_manager.gd`)

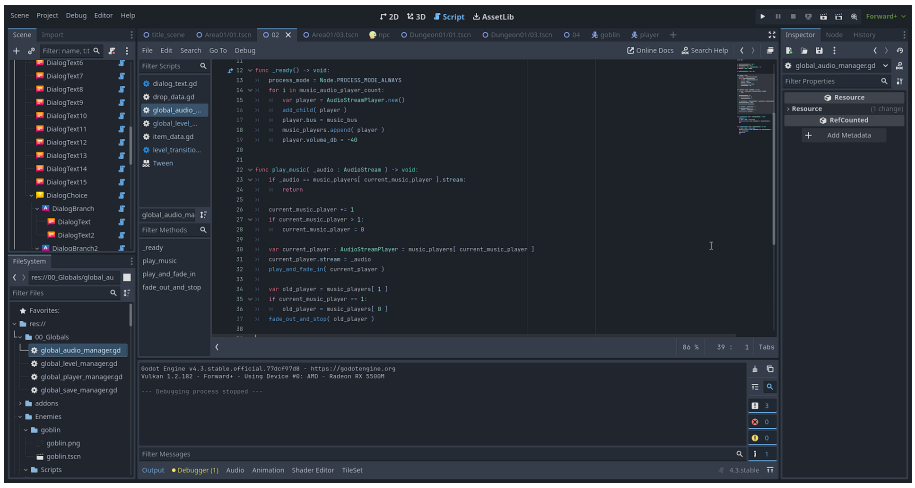

Figure 3.2: Audio Manager

`_ready()`: This method runs when the node is initialized. It sets up `AudioStreamPlayer` nodes (based on `music_audio_player_count`) to handle music playback for each level and sound effects.

```
func _ready() -> void:
    process_mode = Node.PROCESS_MODE_ALWAYS
    for i in music_audio_player_count:
        var player = AudioStreamPlayer.new()
        add_child( player )
        player.bus = music_bus
        music_players.append( player )
        player.volume_db = -40
```

- `AudioStreamPlayer` nodes are added as children of the main node.
- Each player is set to the `music_bus` audio channel.
- Initial volume is reduced to -40 dB (muted).

`play_music()`:

```
func play_music( _audio : AudioStream ) -> void:
        if _audio == music_players[ current_music_player ].stream:
            return

        current_music_player += 1
        if current_music_player > 1:
            current_music_player = 0

        var current_player = music_players[ current_music_player ]
        current_player.stream = _audio
        play_and_fade_in( current_player )

        var old_player = music_players[ 1 ]
        if current_music_player == 1:
            old_player = music_players[ 0 ]
        fade_out_and_stop( old_player )
```

(a) Checks if the requested track is already playing. If yes, it exits without any action.

(b) Alternates between two `AudioStreamPlayer` instances (tracked by `current_music_player`

(c) Assigns the new track to the selected player and starts the fade-in using `play_and_fade_in()`.

(d) Fades out and stops the old track using `fade_out_and_stop()`.

2. Level Manager (`global_level_manager.gd`)

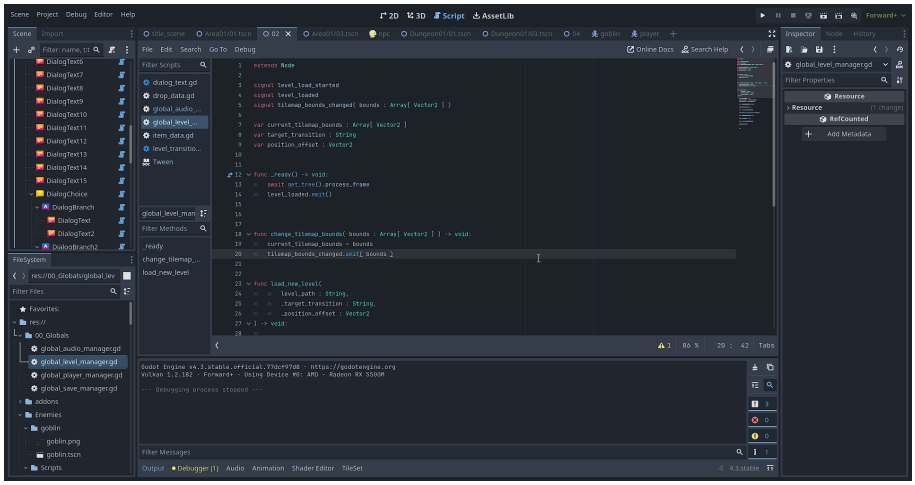

Figure 3.3: Level Manager Scripts

This script uses signals and asynchronous programming (via await) to:

- Ensure safe scene transitions.
- Inform other components about changes using signals (e.g., level loading or boundary updates).
  - `level_load_started`: Emitted when the level loading begins.
  - `level_loaded`: Emitted when the level loading completes.
  - `tilemap_bounds_changed(bounds: Array[Vector2])`: Emitted when the boundaries of the tilemap are updated.

This code relies on the Singleton pattern and incorporates aspects of the Observer pattern.

Important Variables:

- `current_tilemap_bounds: Array[Vector2]`: Stores the current boundaries of the tilemap as an array of vectors. Used to track and update the playable area.

- `target_transition`: String Holds the identifier for the transition or entry point into the new level.

Important Methods:

- `_ready()`: Runs when the node is initialized. Pauses for a frame using await `get_tree().process_frame` to ensure everything is loaded correctly. Emits the `level_loaded signal` to indicate the current level is ready.
- `change_tilemap_bounds(bounds: Array[Vector2])`: Updates the current tilemap boundaries and emits the `tilemap_bounds_changed` signal with the new bounds as a parameter.
- `load_new_level(params)`: Manages the process of transitioning to a new level.
  (a) Pauses the game: Prevents any gameplay during the transition by setting `get_tree().paused = true`.
  (b) Update variables:
    i. `target_transition` is set to the transition target
    ii. `position_offset` adjusts the player's starting position in the new level.
  (c) `SceneTransition.fade_out()` to visually transition to black before loading the next level.
  (d) Emits `level_load_started`
  (e) Processes a frame first to ensure a smooth transition.
  (f) Uses `get_tree().change_scene_to_file(level_path)` to switch to the new level file.
  (g) `SceneTransition.fade_in()` to return from the transition.
  (h) Unpause the game
  (i) Emits `level_loaded` to signal the end of the process.

3. Player Manager (`global_player_manager.gd`)

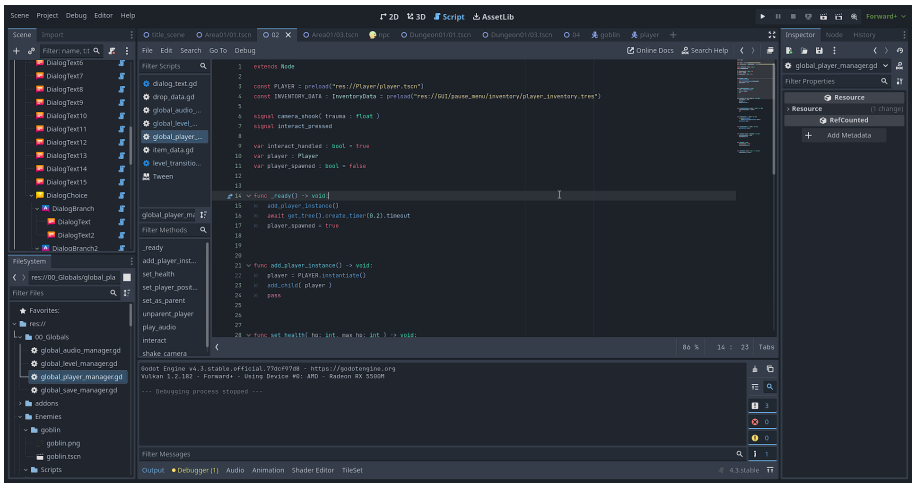

Figure 3.4: Player Manager Script

This script is responsible for managing the player's lifecycle, interaction, health, position, and audio in the game. It includes utility methods for spawning the player, managing their health, triggering interactions, and emitting signals for specific events like camera shaking or interaction. This code relies on the Singleton pattern and incorporates aspects of the Observer pattern.

Signals:

- `interact_pressed`: Emitted when the player interacts with an object. This can be used to trigger interactions with NPCs, items, or environmental elements.

Important Variables:

- `player: Player`: Stores the player instance created from the PLAYER scene. This allows direct access to the player's properties and methods.
- `player_spawned: bool`: Tracks whether the player instance has been successfully spawned in the game.
- `interact_handled: bool`: Indicates whether the interaction input has been processed.

Important Methods:

- `add_player_instance()`: Instantiates the player from the preloaded PLAYER scene and adds it as a child of the current node.

- `set_health(hp: int, max_hp: int)`: Updates the player's health and maximum health values. This method is used for initializing or modifying the player's health during gameplay or after specific events.
- `set_player_position(_new_pos: Vector2)`: Sets the player's global position to a new location.
- `interact()`: Emits the `interact_pressed` signal and sets `interact_handled` to false.

4. Save Manager (`global_save_manager.gd`)

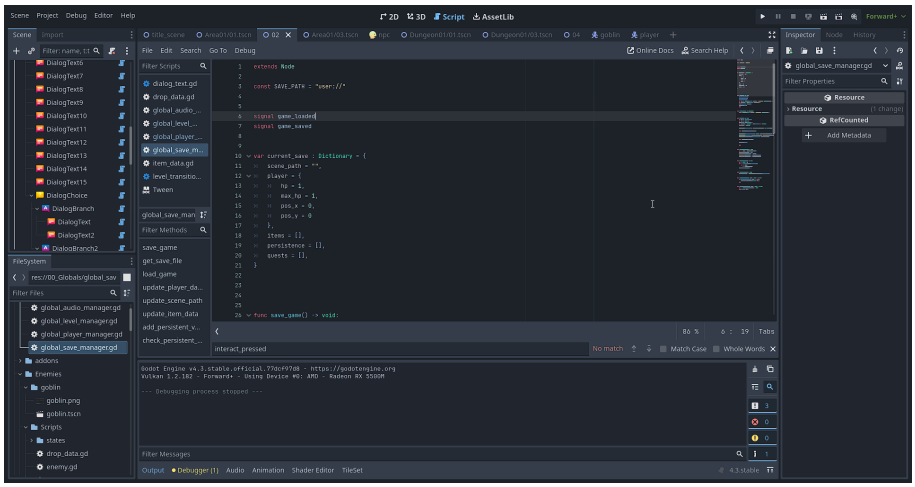

Figure 3.5: Global Save Manager Scripts

The script organizes game state management into:

- Save Game: Updates and stores the player's data, inventory, and scene path in a file.
- Load Game: Reads the saved data, updates the game to match the saved state, and emits relevant signals.
- Persistent Values: Adds or checks values that should remain consistent between sessions.

Signals:

- `game_saved`: Emitted after a game state is saved.
- `game_loaded`: Emitted after the game state is loaded. These signals allow other systems (like a UI) to respond to save/load events.

Important Variables:

- `SAVE_PATH`: The directory where the save file is stored. Defaulted to `user://`
- `current_save`: A dictionary that holds all game state data:
  - Player Info: Health, position.
  - Scene Path: The current level's path.
  - Items and Quests: Inventory and quest progress.
  - Persistence: Values that stay consistent across sessions.

Important Methods:

- `save_game()`:
  (a) Updates player data, current scene, and inventory.
  (b) Saves this information as JSON into a file.
  (c) Emits the `game_saved` signal.
- `load_game()`:
  (a) Reads the save file, parses the JSON data, and updates the game state: (1) Changes the current level using LevelManager.Positions the player and sets their health. (2) Loads inventory data.
  (b) Emits the `game_loaded` signal after completion.
- `update_player_data()`: Extracts player information like health and position from `PlayerManager` and updates the `current_save`.
- `update_scene_path()`: Finds the active level's file path and saves it in `current_save.scene_path`.
- `add_persistent_value(value: String)` and `check_persistent_value(value: String)`: Adds a unique value to the persistence list or checks if it exists.

3.3.2 General Nodes

1. **Enemy Counter**

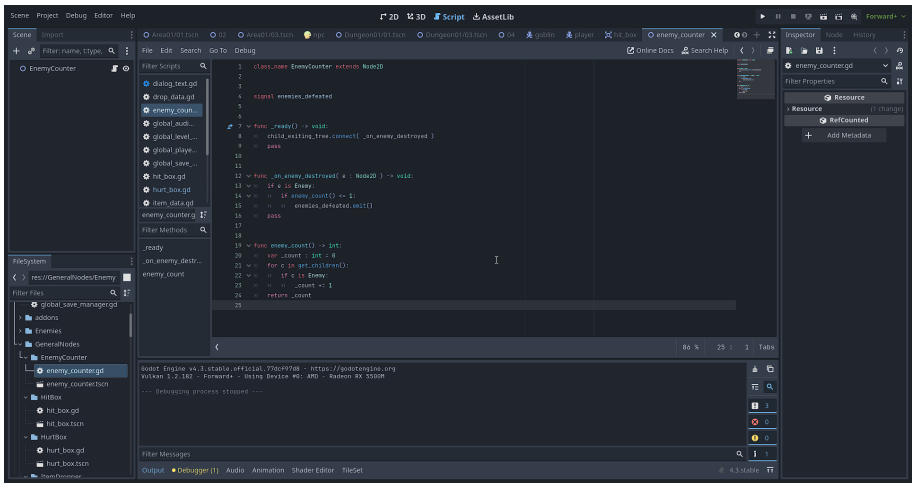

Figure 3.6: Enemy Counter Scripts

```
class_name EnemyCounter extends Node2D

signal enemies_defeated

func _ready() -> void:
 child_exiting_tree.connect( _on_enemy_destroyed )
 pass

func _on_enemy_destroyed( e : Node2D ) -> void:
 if e is Enemy:
  if enemy_count() <= 1:
   enemies_defeated.emit()
 pass

func enemy_count() -> int:
 var _count : int = 0
 for c in get_children():
  if c is Enemy:
   _count += 1
 return _count
```

- Tracks the number of enemies in the scene.
- Emits the `enemies_defeated` signal when all enemies are destroyed.

- Automatically detects and counts child nodes of type `Enemy`.
- Useful for level progression based on enemy elimination.

2. **Hit Box**

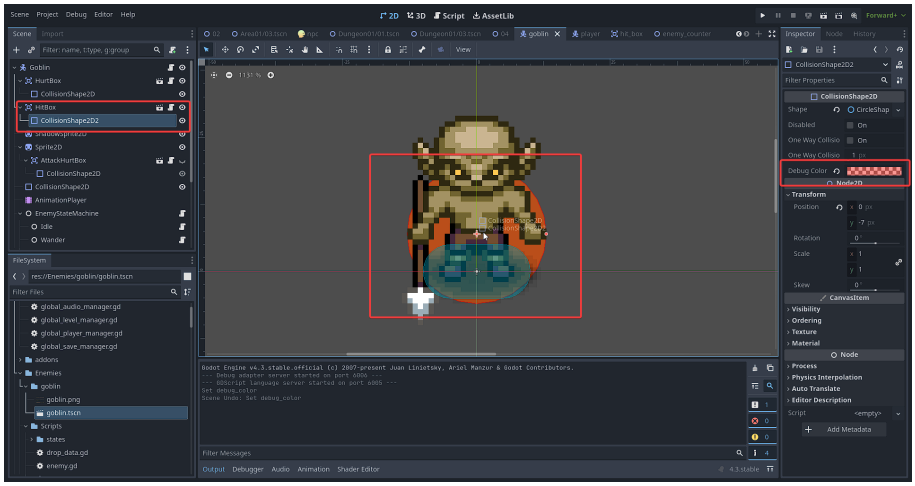

Figure 3.7: Hitbox of Enemy - Goblins

Figure 3.8: Hitbox of Enemy (In game) - Goblins

```
class_name HitBox extends Area2D
signal damaged( hurt_box : HurtBox )

func _ready():
 pass
```

```
func take_damage( hurt_box : HurtBox ) -> void:
 damaged.emit( hurt_box )
```

- Represents a hittable area in the game.
- Emits a damaged signal when it takes damage from a HurtBox.
- Designed to work with combat systems, managing when an entity receives damage.

3. **Hurt Box**

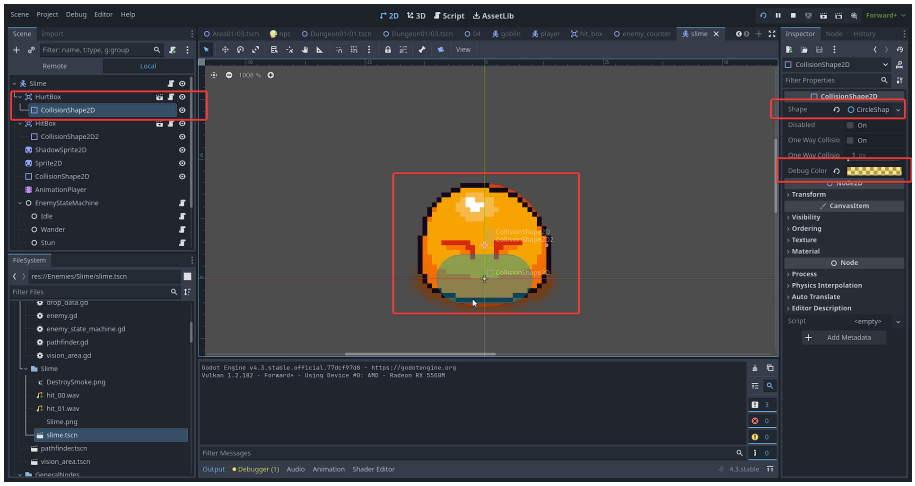

Figure 3.9: Hurt Box of Enemy - Slime

```
class_name HurtBox extends Area2D
signal did_damage
@export var damage : int = 1
func _ready():
 area_entered.connect( _area_entered )
 pass # Replace with function body.

func _area_entered( a : Area2D ) -> void:
 if a is HitBox:
  did_damage.emit()
  a.take_damage( self )
 pass
```

- Represents an area that deals damage when it interacts with a HitBox.
- Emits the `did_damage` signal when it successfully damages a HitBox.
- Contains a damage value and automatically triggers damage upon entering a valid area.

4. **Item Dropper**
   The ItemDropper is a utility node that handles the spawning of items in the game world.

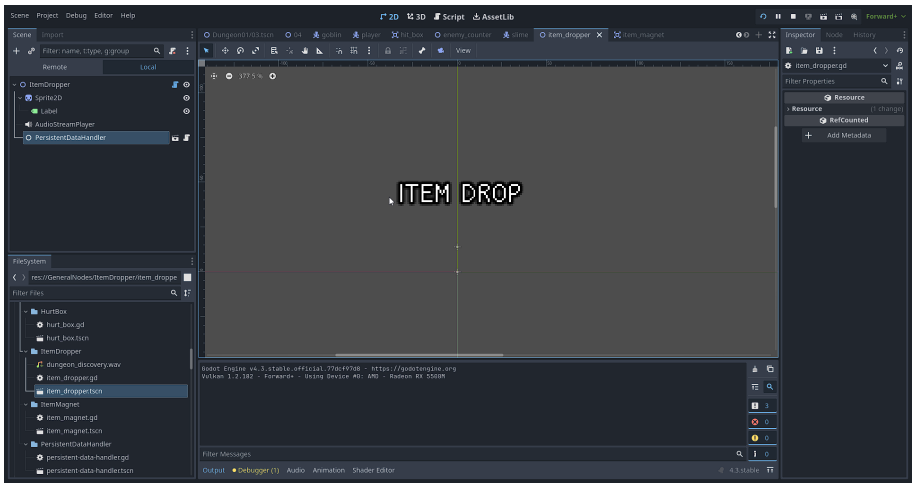

Figure 3.10: Item Dropper Scene

Signals and Variables: None explicitly defined but utilizes connections (e.g., `picked_up`, `data_loaded`) to manage events. `item_data` (exported): Holds data for the item to be dropped, including visual and functional information. `has_dropped`: Tracks if the item has already been dropped.
Important Methods:

- `drop_item()`: Instantiates and drops the item if it hasn't been dropped already.
- `_on_drop_pickup()`: Updates the persistent state when the player collects the item.
- `_on_data_loaded()`: Reads and applies the persistent state to determine if the item should be available for dropping.
- `_set_item_data()`: Updates the item data and ensures the sprite's texture reflects the change in the editor.

5. **Item Magnet**
   Attracts items within its area of effect by applying a force proportional to the `magnet_strength` property. Gradually increase the pull speed until the item reaches the magnet's center.

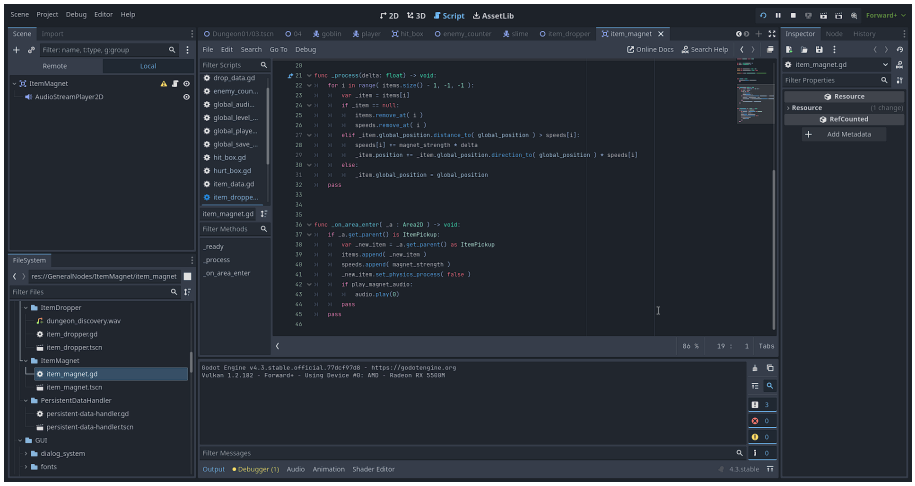

Figure 3.11: Item Magnet

Properties:

- `magnet_strength` (exported): Determines the strength of the pull applied to the items.
- `play_magnet_audio` (exported): Toggles the audio effect for item attraction.

Important Methods:

- `_process(delta: float)`: Iterates over all magnetized items:
  (a) Removes items that no longer exist.
  (b) Gradually moves items closer to the magnet based on their distance and the magnet's strength.
  (c) Finalizes the item's position when it reaches the magnet's center.
- `_on_area_enter(_a: Area2D)`: Adds an item to the magnet's list when it enters the area. Sets the item's physics process to false to allow direct control by the magnet. Plays an audio effect if `play_magnet_audio` is enabled.

3.3.3 Levels

Properties Exported Properties:

level: Scene file path to the next level.

target_transition_area: Name of the transition zone in the target level where the player will appear.

center_player: Boolean to determine whether to center the player in the transition area.

size: Configures the size of the collision area.

side: Enum to specify the transition zone's alignment (LEFT, RIGHT, TOP, BOTTOM).

snap_to_grid: Boolean to snap the transition area's position to the grid.

Other Properties:

collision_shape: Reference to the CollisionShape2D node for dynamically adjusting the collision area. Methods

1. _ready()

Initializes the transition area and ensures proper player placement on level load. Disables monitoring during editor preview.

2. _player_entered(_p: Node2D) Triggers when the player enters the transition area. Calls LevelManager.load_new_level() with parameters like level, target_transition_area, and player offset.

3. _place_player()

Places the player at the correct position within the new level, based on LevelManager.target_transition.

4. get_offset() Calculates the player's offset relative to the transition area, considering the side and center_player settings.

5. _update_area()

Dynamically adjusts the size and position of the collision area (CollisionShape2D) based on the side and size properties.

6. _snap_to_grid()

Snaps the transition area to a grid, aligning it to multiples of 16 pixels. Usage Workflow

Set Up Transition Area:

Attach the LevelTransition node to your scene. Configure the transition area by setting properties like level, side, and size.

Define Target Transition:

Specify the target_transition_area in the next level where the player should appear. Enable Snapping (Optional):

Use snap_to_grid to align the transition area for consistency.

Connect Player Manager:

Ensure PlayerManager and LevelManager are properly integrated to handle player positioning and level loading.

Dependencies:

This script depends on LevelManager and PlayerManager, which are not included in the code. These managers should handle level loading and player positioning. Collision Detection:

Ensure the player character has a CollisionShape2D or CollisionPolygon2D with a matching layer and mask to detect transitions. Grid Alignment:

The grid snapping assumes a grid size of 16x16 pixels. Adjust the logic in _snap_to_grid() if using a different grid size. Editor Compatibility:

The script includes checks for the editor (Engine.is_editor_hint()) to prevent runtime behavior during scene design.

3.3.4 Characters

- Overall this script: Manages the lifecycle of the player character (spawn, parenting, positioning, and health updates). Handles global player-related signals (e.g., camera shake and interaction). Centralizes control over the Player instance for modular and clean code. Signals camera_shook(trauma: float): Emitted when the camera needs to shake. The trauma parameter determines the intensity of the shake. This is useful for adding visual feedback during intense gameplay moments (e.g., taking damage, explosions).

  interact_pressed: Emitted when the player interacts with the environment (e.g., picking up an item, opening a door). The game systems that listen to this signal will determine what happens next.

  Preloaded Resources

  PLAYER: Refers to the player character's scene (res://Player/player.tscn). This is instantiated when the game starts or whenever the player is added to the scene.

  INVENTORY_DATA: References the inventory resource (player_inventory.tres) that likely holds data for the player's inventory (e.g., items, equipment). While not used in this script, it might be shared across different parts of the game (e.g., UI or game logic).

  Variables

interact_handled: Tracks whether the player's interaction (e.g., pressing an interact button) has been handled. This flag can be used to prevent duplicate interactions in a single frame.

player: Stores a reference to the instantiated Player scene.

player_spawned: A boolean flag to indicate whether the player has been fully spawned in the scene. It becomes true after a short delay following instantiation.

Methods

1. Initialization _ready(): Calls add_player_instance() to instantiate and add the Player scene as a child. Uses a short timer (create_timer(0.2)) before setting player_spawned to true. This delay ensures the player is fully initialized before interacting with other game systems.

2. Player Instance Management

add_player_instance(): Instantiates the Player scene and adds it as a child of this node. This ensures the Player is part of the scene tree.

set_as_parent(_p: Node2D): Detaches the player from its current parent and reattaches it to _p. Useful for dynamic parenting during gameplay (e.g., transitioning the player to another area).

unparent_player(_p: Node2D): Removes the player from the provided parent. This complements set_as_parent() for flexible player placement in the scene hierarchy.

3. Player State Updates

set_health(hp: int, max_hp: int): Updates the player's hp and max_hp values. Calls update_hp(0) to refresh the player's health UI or any associated logic.

set_player_position(_new_pos: Vector2): Sets the player's global position. Useful for spawning the player at specific locations (e.g., checkpoints).

4. Audio Playback play_audio(_audio: AudioStream): Assigns an AudioStream to the player's audio player (player.audio) and plays it. This centralizes audio playback for actions associated with the player (e.g., footsteps, attack sounds).

5. Interaction and Camera Shake interact(): Sets interact_handled to false and emits the interact_pressed signal. Other systems listening to this signal can handle the interaction logic. shake_camera(trauma: float = 1):

Emits the camera_shook signal, passing the trauma value. This can trigger camera effects, such as shaking, with varying intensity. Code Highlights Player Instantiation:

The add_player_instance() function provides a reusable method for spawning the player in the scene. It allows the Player to be dynamically added or respawned during gameplay.

Parenting Flexibility: The set_as_parent() and unparent_player() methods give full control over the player's position in the scene hierarchy. This is particularly useful for games with complex interactions, such as cutscenes or area transitions.

Signal-Based Architecture: The use of signals (camera_shook, interact_pressed) allows other game systems to respond to player actions without tightly coupling logic. This makes the code more modular and easier to maintain.

Health Updates: The set_health() method provides a clean way to initialize or modify the player's health while ensuring consistency across systems (e.g., HUD updates).

Error Handling: Ensure the Player scene (res://Player/player.tscn) and player_inventory.tres exist and are correctly implemented. Missing resources will cause runtime errors.

Signals: Systems listening to interact_pressed and camera_shook should handle these signals properly to avoid unintended behavior.

Interaction Logic: The interact_handled flag is set to false in interact(), but there's no code to reset it to true. Ensure that interactions properly manage this flag to avoid potential issues.

Audio Playback: If multiple audio effects overlap, consider whether the player.audio node should use a queue system or limit simultaneous playback.

Camera Shake: Implement the camera shake logic in the system connected to the camera_shook signal. The intensity (trauma) should scale the visual effect appropriately.

- Player: This Player class extends CharacterBody2D and includes core features such as:

  Directional movement and animations. Health management with invulnerability mechanics. State machine integration for modular behavior. Signals for interaction with external systems (e.g., damage handling, directional changes). Item pickup and revival functionalities. Signals direction_changed(new_direction: Vector2): Emitted when the player's cardinal direction changes.

  player_damaged(hurt_box: HurtBox): Emitted when the player takes damage, passing the HurtBox instance that caused the damage.

Constants DIR_4: Defines the four cardinal directions (RIGHT, DOWN, LEFT, UP) for movement and animations. Properties Movement & Direction:

cardinal_direction: The current facing direction of the player (e.g., Vector2.UP). direction: Normalized vector indicating movement input. Health Management:

hp: Current health points. max_hp: Maximum health points. invulnerable: Boolean to track if the player is temporarily immune to damage. References to Child Nodes:

animation_player, effect_animation_player: Handles animations for the player. hit_box: Represents the player's collision area for damage detection. state_machine: Manages the player's current state. audio: Plays audio effects. lift and carry: States for lifting and carrying objects. held_item: Represents the currently held item, if any. Main Methods 1. Movement _process(_delta):

Captures input from Input.get_axis and calculates the movement direction. Normalizes the direction for consistent movement speed. _physics_process(_delta):

Applies movement to the player using move_and_slide(). set_direction():

Updates the player's cardinal_direction based on the input direction. Emits the direction_changed signal if the direction has changed. Adjusts the player's sprite scale for horizontal flipping (left/right). 2. Animation update_animation(state: String):

Plays the appropriate animation based on the player's state and direction. anim_direction():

Maps the cardinal_direction to a string ("up", "down", or "side") for use in animation names. 3. Health and Damage _take_damage(hurt_box: HurtBox):

Reduces the player's HP if not invulnerable and emits the player_damaged signal. update_hp(delta: int):

Updates the player's health points and clamps the value between 0 and max_hp. Notifies the HUD (PlayerHud.update_hp) to visually reflect the change. make_invulnerable(_dura
float = 1.0):

Temporarily sets the player to an invulnerable state and disables hitbox monitoring. Re-enables damage detection after the specified duration. revive_player():

Fully restores the player's HP and switches to the idle state. 4. Item Interaction pickup_item(_t: Throwable): Switches the player's state to lift and assigns the picked-up item to the carry state. Initialization _ready(): Sets up the player within the PlayerManager for global access. Initializes the state_machine with the player as its context. Connects the hit_box.damaged signal to the _take_damage() method.

Updates the player's HP to full (default value: 99). Gameplay Integration State Machine:

Modularizes player behavior into different states (Idle, Lift, Carry, etc.). The state machine logic is managed separately in the PlayerStateMachine node. Damage Handling:

Integrates with external systems through the HurtBox and player_damaged signal. Allows temporary immunity via make_invulnerable(). HUD Updates:

Uses the PlayerHud class to synchronize visual HP changes with the player's health status. Item Pickup:

Supports carrying throwable items, enhancing interaction and gameplay mechanics.

3.3.5 Dialog System

The DialogSystemNode is a Godot-based dialog system built to manage dynamic, interactive conversations in a game. It displays character dialog, manages text effects like typing, and supports branching choices, creating immersive storytelling experiences.

Signals

finished: Triggered when the dialog sequence ends.

letter_added(letter: String): Triggered as each letter is typed for additional effects (e.g., sound, animation).

Properties

text_speed: Controls the typing speed of the dialog.

is_active, text_in_progress, and waiting_for_choice: Flags for the system's state during interactions.

3.3.6 Further Development

Strength: In this project, we implemented many game materials such as: teleport gate, game collision, game NPCs, boss, items, etc. We already create some game maps, game levels and basic data structure and algorithm quiz. We also prepare many other game assets for further development.

Weakness: This project focus on create many materials for further development from scratch, so there was not so much game mechanism for player. In future, we will add more mini game during the dungeon adventure.

4 Summary

This game assignment project focuses on developing a robust Player Management System using the Godot Engine. The system handles player instantiation, health management, interaction mechanics, and dynamic re-parenting for versatile gameplay scenarios. Core features include a signal-based interaction system for modular and seamless object interactions, a centralized health update mechanism, and immersive camera shake effects triggered by gameplay events. Additionally, the system supports audio playback for player actions and integrates an inventory resource for future inventory management. With its modular design and scalability, this project provides a solid foundation for building complex gameplay systems and enhancing player experience.