Mohammad Ibrahim Salman, April 2025

Final Year Project Report

Full Unit – Final Report

Building a Game

Mohammad Ibrahim Salman

Final Report submitted in part fulfilment of the degree of
BSc (Hons) in Computer Science

Supervisor: Sourjah, Susnas

Department of Computer Science Royal Holloway,
University of London

Egham, April 2025

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 28,672

Student Name: Mohammad Ibrahim Salman

Date of Submission: 29/04/2025

Signature: Mohammad Ibrahim Salman

Table of Content

1. Abstract

This report presents the design and development of *Wimbledon's Lot*, a top-down 2D retro-style game built using the Godot engine. Centered around decision-making, exploration, and narrative immersion, the game integrates key game design patterns such as ECS (Entity-Component-System), Finite State Machines, and Observer Patterns. These patterns, combined with theoretical frameworks like Flow Theory and Bartle's Taxonomy, shape a rich and engaging player experience. Development followed an iterative, modular approach, employing version control through Git and structured documentation via a comprehensive Game Design Document (GDD).

The game world—centered on the town of Castle Rock—was built using Godot's Tile Map and scene system, with open-source assets and carefully layered collision structures to support smooth gameplay. Core mechanics, including player movement, animation switching, and terrain interaction, were developed with attention to detail, while more advanced systems such as inventory management and item pickups were incrementally integrated. Testing, code quality practices, and requirement elicitation methods reflect principles of software engineering, ensuring a robust and extensible codebase.

This report documents the complete journey of the project, reflecting both technical achievements and personal growth throughout the development process. It provides a critical analysis of the project's progress, detailing specific implementations, iterative refinements, and the error handling strategies employed to overcome technical challenges such as asset management conflicts, design bottlenecks, and bugs. Lessons learned are discussed alongside adaptive problem-solving approaches, demonstrating technical proficiency and reflective practice. In addition to showcasing the evolving development through a detailed diary, the report examines professional issues such as open licensing, version

control, and platform compatibility, offering a comprehensive view of the project's scope, evolution, and potential future enhancements.

2. Introduction

The development of video games is a complex and multifaceted process, requiring the integration of creative vision, technical implementation, and user experience design. *Wimbledon's Lot* is a top-down 2D retro-style game developed using the Godot Engine, aimed at blending the nostalgic aesthetic of the 16- and 32-bit gaming eras with modern game design sensibilities. The project centers around exploration, decision-making, and narrative immersion, with a strong emphasis on player agency and strategic gameplay.

This report presents the planning, development, and refinement process of *Wimbledon's Lot*, reflecting both technical achievements and personal growth. It critically analyzes the challenges encountered during implementation, the methodologies applied, and the lessons learned throughout the project lifecycle. In doing so, it aims to contribute not only a playable game prototype but also a structured case study in independent game development.

The vision behind *Wimbledon's Lot* was to create a rich, expansive adventure experience that captures the spirit of classic gaming while introducing modern game design principles. The current build serves as a prototype or demo, providing a working showcase of the game's core mechanics, environment design, and artistic direction. The initial level layout demonstrates how exploration, player movement, inventory management, and environmental interactions are intended to function. This foundational work establishes a scalable framework upon which additional gameplay features, narrative depth, and extended content could be built, paving the way for *Wimbledon's Lot* to evolve into a complete and fully developed game in the future.

2.1 Aims & Objectives

The primary aim of this project is to design and implement a playable, immersive 2D game that demonstrates strong technical design, thoughtful narrative integration, and high-quality gameplay mechanics. The objectives of the project are as follows:

- To create a top-down 2D adventure game incorporating exploration, interaction, and combat mechanics.
- To utilize open-source technologies, primarily the Godot Engine, for game development.
- To apply design patterns such as Entity-Component-System (ECS), Finite State Machines (FSM), and Observer Patterns to promote modularity and scalability.
- To develop a dynamic game environment through the use of Godot's node-based architecture, tile-maps, and scene systems.
- To implement a responsive player movement system, inventory management system, and basic enemy interaction mechanics.
- To document the development process comprehensively, reflecting on technical challenges, iterative refinement, and professional issues encountered.
- To demonstrate key software engineering practices, including version control, requirement elicitation, testing, and code quality assurance.
- To explore professional considerations such as licensing, platform compatibility, and asset management.

2.2 Motivations

The motivation for this project is rooted in the dynamic and rapidly evolving nature of the video game industry, a sector that has become a cornerstone of global entertainment. With annual revenues surpassing those of the film and music industries combined, the gaming sector offers immense opportunities alongside significant challenges. One of the key drivers of growth has been the shift toward digital distribution, transforming how games are marketed, sold, and consumed. Platforms like Steam, the Epic Games Store, and mobile app marketplaces have opened new pathways for independent developers to reach global audiences. However, this democratization has also led to market saturation, resulting in a flood of low-quality, repetitive, and overly simplistic free-to-play games [4].

Wimbledon's Lot seeks to stand apart by delivering a free-to-play experience that avoids these common pitfalls. As a first-time programmer, my ambition was to demonstrate that, even with limited resources, it is possible to create a meaningful and engaging game. Instead of relying on intrusive monetization models like excessive in-app purchases or advertising, the project focuses on player agency, strategic decision-making, exploration, and narrative immersion. The goal was to create an experience that rewards thoughtful play and emotional investment, countering the assumption that free games must be shallow or uninspired [5].

Another key motivation was the exploration of the economic and technological trends shaping the industry. By examining innovations such as social media integration, in-game advertising, freemium pricing models, and cloud gaming, I sought to better understand how these forces influence both developers and consumers. This research not only informed design decisions for *Wimbledon's Lot* but also provided insights into how independent games can create sustainable value in a crowded marketplace [4].

Theoretical frameworks also played a critical role in shaping the project. Flow Theory, developed by Mihaly Csikszentmihalyi, emphasizes the importance of maintaining a balance between challenge and skill to immerse players in a state of "flow." This principle informed the game's gradual difficulty progression, ensuring that players are continually engaged without feeling overwhelmed [1]. Bartle's Taxonomy of Player Types further guided the design by recognizing the diverse motivations of players — Achievers, Explorers, Socializers, and Killers — and building gameplay elements that appeal to each type [2]. Environmental

Storytelling Theory provided the foundation for embedding narrative clues within the game's spaces, encouraging players to discover the story through exploration rather than exposition [3].

On a personal level, my passion for video games and game development has been a long-standing one. I have always followed the latest industry news, from new engine releases and modifications to evolving design philosophies and business models. However, when starting this project, I went into the development process almost completely blind, with little practical knowledge of how to actually build a game. Choosing Godot as the development engine proved to be a pivotal decision; its beginner-friendly architecture, Python-like GDScript, and supportive community made it an ideal platform for someone taking their first steps into game development. Through trial, error, and persistent learning, I transformed an initial vision into a working prototype — an experience that not only deepened my technical skills but also strengthened my appreciation for the craft of game development.

Ultimately, *Wimbledon's Lot* is more than a technical project; it represents a personal journey of growth, creativity, and resilience. It demonstrates that with careful planning, a strong foundation in theoretical frameworks, and a willingness to embrace challenges, even first-time developers can create experiences that are rich, engaging, and full of potential.

2.3 Key Milestones Achieved

The development of *Wimbledon's Lot* followed a structured and iterative process, with major milestones achieved throughout the project timeline. These milestones mark critical phases in the game's development, reflecting both technical progress and personal growth.

- **05/11/2024 – Initial Planning and GDD Drafting**

  Began drafting the Game Design Document (GDD), outlining the project's vision, gameplay mechanics, story, environment, and user interface. Simultaneously imported a large set of open-source assets and organized them for future use

- **12/11/2024 to 13/11/2024 – Completion of the GDD**

  Fully completed all GDD sections, including Character, Narrative, Technical, and Marketing details, providing a clear development roadmap

- **17/11/2024 – Initial Character Development**

  Created the first version of the playable character, complete with idle animations and collision setup. Solved major node-structure bugs affecting movement

- **26/11/2024 – Advanced Player Movement and Animation**

  *Implemented smooth Zelda-style movement with full directional animation switching. Integrated Godot's AnimationPlayer for seamless transitions between idle and running state*s

- **16/02/2025 to 23/02/2025 – Terrain and Overworld Creation**

  Began constructing the overworld, creating and painting terrain using Godot's TileMap and TileSet systems. Introduced paths, lakes, and other terrain features to bring the environment to life

- **26/02/2025 to 01/03/2025 – Verticality and Hub World Creation**

  Designed cliff terrains, rivers, and bridges, layering multiple TileMaps for decorative and navigational elements. Established the initial hub area layout where gameplay begins

- **06/03/2025 – Environmental Detailing and World Boundaries**

  Completed the foundational hub world, decorating it with trees, rocks, and water features while beginning the implementation of invisible collision boundaries to contain player movement

- **11/03/2025 to 19/03/2025 – Major Technical Challenges Overcome**

  Addressed significant setbacks, including scene corruption due to Git mismanagement, *lost decorations, and file corruption. Successfully debugged and recovered the overworld by manual file editing and redecoration*

- **19/03/2025 – Camera System and Collision Implementation**

  Implemented a Camera2D system to follow the player and improved collision layers to ensure proper interaction with environmental elements like water, cliffs, and bridges

- **21/03/2025 to 24/03/2025 – Pickup System and Early Inventory Prototype**

  Developed the item pickup system using Area2D detection and connected it to an initial inventory backend. Introduced a system for dynamically loading pickups and updating item quantities

- **06/04/2025 to 10/04/2025 – Inventory UI Design and Dynamic Slot System**

  Created a modular inventory UI using GridContainer and CanvasLayer structures. Implemented dynamic generation of inventory slots and connected UI elements to inventory backend scripts

- **10/04/2025 – Stackable Items and Inventory Data Management**

  Expanded inventory functionality to handle stackable items like gold coins, including logic for stacking, splitting, and displaying item quantities within slots

3. Background Theory

3.1 Literature Review

3.1.1 Relevant to the Project

The design and development of video games require the integration of theories from psychology, sociology, technology, and software engineering. This section reviews the key theories, technological tools, and professional insights that shaped the development of *Wimbledon's Lot*. The insights drawn from both academic sources and professional documentation guided the gameplay design, player interaction models, and technical implementation strategies.

3.1.2 Theoretical Foundations of Gameplay Design

Mihaly Csikszentmihalyi's Flow Theory provides one of the most fundamental psychological frameworks applied in game design. Flow describes the mental state where an individual is fully immersed in a task, experiencing energized focus, deep involvement, and enjoyment [1]. Maintaining flow in gameplay requires a careful balance between the player's skills and the game's challenges. In *Wimbledon's Lot*, this concept informed the gradual scaling of difficulty, ensuring players are neither overwhelmed nor bored as they progress.

Bartle's Taxonomy of Player Types categorizes players into four groups: Achievers, Explorers, Socializers, and Killers [2]. Understanding this taxonomy allowed the design of *Wimbledon's Lot* to appeal to a broader audience by embedding different styles of engagement: Achievers are rewarded through quest completion, Explorers discover environmental lore, Socializers interact through branching dialogues with NPCs, and Killers are challenged through combat encounters. Integrating Bartle's framework ensured that the gameplay experience catered to varying player motivations, enriching player satisfaction and retention.

Environmental Storytelling Theory, as outlined by Jenkins (2004), emphasizes the use of physical space and environmental cues to convey narrative without relying heavily on dialogue or exposition [3]. In *Wimbledon's Lot*, this theory shaped the level designs, where objects, relics, and environmental changes subtly hint at deeper lore, encouraging exploration and discovery.

3.1.3 Industry Context and Market Trends

The broader industry context also influenced the project's design philosophy. Marchand and Hennig-Thurau (2013) examined the shifts in the gaming market, noting how digital distribution platforms, social media integration, and free-to-play business models have reshaped developer strategies [4]. Their findings provided crucial insights into avoiding the pitfalls of "low-quality" free-to-play games that flood the market, particularly in mobile and indie spaces. By focusing on player agency and engagement rather than monetization, *Wimbledon's Lot* seeks to deliver a richer experience in a crowded marketplace [4].

Furthermore, the economic viability of emerging models such as freemium pricing, cloud gaming, and subscription services was analyzed. Hamari and Järvinen (2011) explored the effectiveness of these models in sustaining independent game development [5]. Understanding these trends shaped the decision to structure *Wimbledon's Lot* as a complete experience without microtransactions or paywalls, aligning with best practices for player trust and satisfaction.

3.1.4 Technical Implementation and Tools

Technological decisions were heavily informed by both academic research and professional documentation. Godot Engine was chosen due to its open-source nature, accessibility for beginners, and robust documentation [17]. Godot's node and scene architecture promotes modular and scalable design patterns, crucial for managing the complexity of game development. The official Godot documentation provided key guidance on concepts like Singleton usage (Autoloads), input handling with InputEvent, and the use of signals for scene communication, all of which were extensively applied during the project [17].

Singletons were used to manage global game states, enabling efficient data persistence across scenes. The InputEvent system was implemented to refine player controls and handle complex input actions cleanly, ensuring a smoother gameplay experience. Additionally, the Godot documentation's emphasis on modular scene design aligned closely with the Entity-Component-System (ECS) architectural patterns adopted in the project [17].

Furthermore, the practical application of design patterns such as Finite State Machines and Observer Patterns was informed by academic and industrial best practices. For example, behavioral states for characters and interactive objects were structured using finite state machines, ensuring organized, scalable code and clear state transitions [6][7].

The importance of thorough testing and requirement elicitation was emphasized by Sommerville (2016), who outlines the need for systematic approaches to software validation and verification [8]. While informal, the development diary acted as a lightweight requirements traceability document, mapping planned features to their actual implementation, and testing sessions iteratively improved the game's robustness.

3.1.5 Player Interaction and User Experience

Player interaction models and user experience (UX) theories also informed design decisions. Norman's Affordance Theory highlights that interface elements should visually suggest their usage [10]. This theory influenced the design of *Wimbledon's Lot*'s UI elements, ensuring buttons, dialogue boxes, and inventory slots were intuitive and self-explanatory.

Csikszentmihalyi's Flow concepts were reinforced through level pacing and environmental challenges, maintaining immersion while avoiding cognitive overload [1]. Environmental storytelling techniques were deeply integrated into the world-building, enhancing immersion through visual and interactive narrative delivery [3].

Finally, insights from game design authors such as Jesse Schell [11] and Tracy Fullerton [12] provided further professional strategies, particularly in areas of system thinking, iteration, and player-centered design, ensuring that every feature added to *Wimbledon's Lot* served a purpose and enriched the player's journey.

3. 2 Theoretical Application to the Project

The theoretical frameworks reviewed previously were not only studied for academic completeness but actively influenced the practical design and development of *Wimbledon's Lot*. Throughout the project, theories were applied in an intentional and structured way to ensure the gameplay experience was engaging, immersive, and robust.

3.2.1 Flow Theory in Gameplay Progression

Flow Theory, developed by Csikszentmihalyi, emphasizes maintaining an optimal balance between challenge and player skill to achieve deep engagement [1]. In *Wimbledon's Lot*, this theory guided the incremental increase of gameplay difficulty.

During the early development stages (November 2024), the initial level was designed to introduce basic mechanics such as movement, item collection, and environment interaction at a manageable pace. As players become comfortable with controls, environmental complexity increases — for example, introducing blocked paths requiring exploration, or hidden pickups, thereby escalating difficulty in line with player growth. Care was taken to avoid overwhelming new players; collision boundaries and clear environmental hints (such as decorative pathways and cliff guides) ensured that players always had a sense of direction without feeling frustrated.

3.2.2 Bartle's Taxonomy in Player Motivation Design

Bartle's Taxonomy [2] was directly mapped into the design of the hub world and planned future levels.

- **Achievers** are incentivized through clear objectives like item collection quests and potential hidden achievements.

- **Explorers** are rewarded by discovering environmental storytelling elements such as journal entries, relics, or secret pathways embedded naturally into the world's design.
- **Socializers** are introduced through interactive NPCs (planned), featuring branching dialogue options that encourage player engagement beyond mere exploration.
- **Killers** will eventually be challenged through enemy encounters; early collision and player movement systems were built with future combat systems in mind.

Integrating Bartle's taxonomy ensured the game design catered to multiple playstyles, thereby broadening the appeal of *Wimbledon's Lot*.

3.2.3 Environmental Storytelling in Level Design

Environmental Storytelling Theory [3] influenced every stage of overworld design. Rather than relying on explicit exposition, the narrative was embedded into the physical world through:

- Broken structures implying past conflicts,
- Environmental cues hinting at hidden histories (like ruins, abandoned homes),
- placement of natural obstacles like cliffs and rivers that suggest the world's "lived-in" history.

Designing with environmental storytelling in mind allowed players to infer the backstory by exploration, making the world feel richer and more dynamic even at the prototype stage.

3.2.4 Technical Frameworks and Best Practices

The Godot Engine's official documentation [17] was an indispensable technical resource. Several key concepts were practically applied:

- **Singletons (Autoloads)** were used to manage global player data, inventory items, and overarching game states across multiple scenes.

- **InputEvent System** was used to implement smooth player movement and interaction controls.
- **Signals and Scene Communication** were heavily utilized in connecting UI elements (like inventory slots) to player actions and environmental objects.

These practices promoted a clean, modular architecture, which was essential for managing the increasing complexity of the game as it developed.

Furthermore, Entity-Component-System (ECS) principles were partially implemented by using Godot’s node and scene system to separate concerns between movement logic, animation control, collision detection, and world interaction.

The use of Finite State Machines for player state transitions (Idle → Running → Attacking) was planned into the architecture, even if not fully realized at the current demo stage.

3.2.5 Software Engineering Methodologies

Software engineering principles, as outlined by Sommerville [8], were applied in a lightweight yet effective manner.

- **Version Control (Git)** was employed throughout development to manage changes, although early branching strategies led to some significant challenges and data loss
- **Requirement Elicitation** was conducted informally through diary entries, mapping goals like "smooth movement," "interactive inventory," and "environmental storytelling" to tangible technical tasks.
- **Testing and Validation** were integrated iteratively: after each feature implementation (e.g., player movement, item pickup), testing was conducted manually to identify bugs (such as collision issues or item pickup failures) and appropriate fixes were deployed.

By maintaining flexibility while adhering to key engineering best practices, development was kept structured and recoverable even in the face of unexpected setbacks.

4. Project Development and Methodology

4.1 Technology Choices

The development of *Wimbledon's Lot* was supported by a range of carefully selected technologies, each chosen for its suitability to the project's scope, my skill level as a first-time developer, and the project's objectives.

4.1.1 Game Engine: Godot Engine (v4.2 Stable)

Godot was chosen as the primary development platform due to its open-source nature, intuitive node-based architecture, and ease of use for beginners [17]. Its documentation provided critical support for learning core concepts such as scene instancing, Singleton patterns (Autoload), and signal-based communication. Godot's lightweight system requirements also allowed development on modest hardware while offering powerful 2D capabilities.

4.1.2 Git Bash and GitHub

Git Bash was used locally to manage version control via Git, while GitHub hosted the project's repository remotely. This setup allowed for backups, branch management, and collaborative workflow simulation, even as a solo developer.
GitHub also served as an essential tool for maintaining a historical record of development changes, an important principle in software engineering [8].

4.1.3 Open-Source Asset Libraries

Assets for tilesets, character sprites, and environmental decorations were sourced from sites such as OpenGameArt.org and itch.io. Using open-licensed content allowed rapid prototyping while ensuring legal compliance and artistic consistency [14].

4.1.4 Godot Official Documentation

The Godot Engine's official documentation [17] was vital for technical reference during implementation. Concepts such as input handling (InputEvent), physics integration, and UI design were directly applied.

4.1.5 Development Hardware

Development was carried out on a laptop with the following specifications:

- **Device Name**: DESKTOP-06ASFD9
- **Processor**: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
- **RAM**: 8.00 GB
- **System Type**: 64-bit Operating System, x64-based processor
- **OS**: Windows 11 Home, Version 23H2, OS Build 22631.5262
- **Input**: Pen and touch support (10 points)

The device specifications provided sufficient resources for 2D game development, albeit with occasional performance bottlenecks during larger scene renders.

4.2 Implementation So Far

The development progress can be broken into several key systems and gameplay features.

4.2.1 Game Design Document (GDD) Completion

A detailed GDD was produced early in the project, covering character design, narrative structure, gameplay systems, technical requirements, and marketing considerations. This document served as a living blueprint throughout development.

4.2.2 World Building and Level Design

The hub world, Castle Rock, was created using Godot's TileMap system. Layered tilesets provided verticality, paths, water features, and decorative structures, creating an immersive starting area.

4.2.3 Player Movement System

Smooth, eight-directional player movement was implemented using InputEvent systems and Godot’s built-in physics. Collision handling was refined to prevent the player from walking through obstacles such as cliffs and water bodies.

4.2.4 Camera System

A Camera2D was configured to follow the player dynamically across the environment, enhancing player immersion.

4.2.5 Pickup System and Inventory Backend

Area2D-based pickups allow players to collect items scattered throughout the environment. An early inventory system tracks collected items using a Singleton data model, allowing persistence between scenes.

4.2.6 User Interface (UI) Design

An inventory UI was designed using Godot’s GridContainer system. Items are dynamically displayed in slots, supporting stackable items and interactive feedback.

4.2.7 Basic Save and Load Systems (Prototype Stage)

Preliminary work began on systems to persist player progress between sessions, although full save/load functionality is earmarked for future development.

4.3 Critical Analysis and Discussion

The development of *Wimbledon's Lot* was a transformative yet challenging journey, filled with valuable lessons about technical development, project management, and personal resilience. Early plans laid out in the Project Planset ambitious goals, such as implementing advanced AI, complex puzzle logic, save/load systems, and even online features. However, the realities of independent development — including technical complexity, time constraints, and personal workload from other demanding modules like Deep Learning and Digital Forensics — necessitated major compromises.

The development process itself is inherently solitary, even when working in a team setting. Much of the work involves long hours dedicated to a computer, debugging issues that sometimes take days to resolve. This "anti-social" aspect of programming can lead to physical fatigue and psychological stress, including imposter syndrome — a feeling that no amount of progress meets the milestones initially set.

One of the most significant compromises involved the abandonment of full AI implementation. Enemy tracking and dynamic behavior systems were initially planned but proved too complex within the available timeframe and hardware limitations. Instead, a manually controlled movement system was implemented, with enemies following simple, predefined patterns. Similarly, systems like online leaderboards and full multiplayer integration, which were ambitious parts of the initial roadmap, were deferred for potential future work.

This experience mirrors industry challenges on a larger scale. For instance, CD Projekt's *Cyberpunk 2077* serves as a relevant case study. The game promised revolutionary RPG mechanics, deep AI-driven decision-making, and high-fidelity graphics — but at launch, many of these systems were incomplete or missing. Core features like advanced AI, dynamic police systems, and property acquisition were ultimately scrapped or heavily delayed. Post-launch, the company had to pivot its marketing and gameplay focus from a full RPG to a more traditional action-adventure structure, a situation well-documented across industry media [18]. Like *Cyberpunk 2077*, *Wimbledon's Lot* underwent a scope reduction based on practical limitations, highlighting how even large-scale projects often cannot meet all initial objectives.

Technical choices made early on — such as selecting the Godot Engine — greatly influenced the project's success. Godot’s strong documentation [17] and open-source flexibility provided critical support. Its use of node-based architecture and scripting through GDScript allowed rapid development and experimentation, aligning with advice from Zechner and Green in *Beginning Android Games*, who advocate for using lightweight, modular engines in early game development [15].

Similarly, applying game programming patterns from Nystrom’s *Game Programming Patterns* [16] (such as the Singleton pattern and event-based programming) enabled efficient architecture design. These patterns simplified complex interactions between systems like inventory management, player input, and UI rendering.

Despite the learning curve, Git Bash and GitHub proved to be indispensable tools for version control. Nevertheless, early missteps — such as scene corruption due to improper branching — highlighted the importance of more rigorous repository management. As Zechner and Green [15] suggest, maintaining modularity and backup strategies is crucial, even in small game projects.

Overall, the experience reaffirmed that successful game development requires not only technical skill but also adaptability, time management, and a willingness to iterate on initial ideas. In hindsight, project scoping should have been more conservative, aligning better with available time and resources. Rather than viewing these compromises as failures, they reflect a maturing understanding of the complex, iterative nature of real-world software and game development.

Ultimately, the project demonstrates that passion, planning, and practical application of theory can lead to meaningful progress — even if the final product differs from the original vision. It shows that creative endeavors like game development are not about perfectly replicating plans, but about learning, adapting, and creating something unique within the limits imposed by reality.

5. Software Engineering

5.1 Requirement Elicitation

The requirements for *Wimbledon's Lot* were defined both at the project's outset and refined iteratively during development. They fall into two categories:

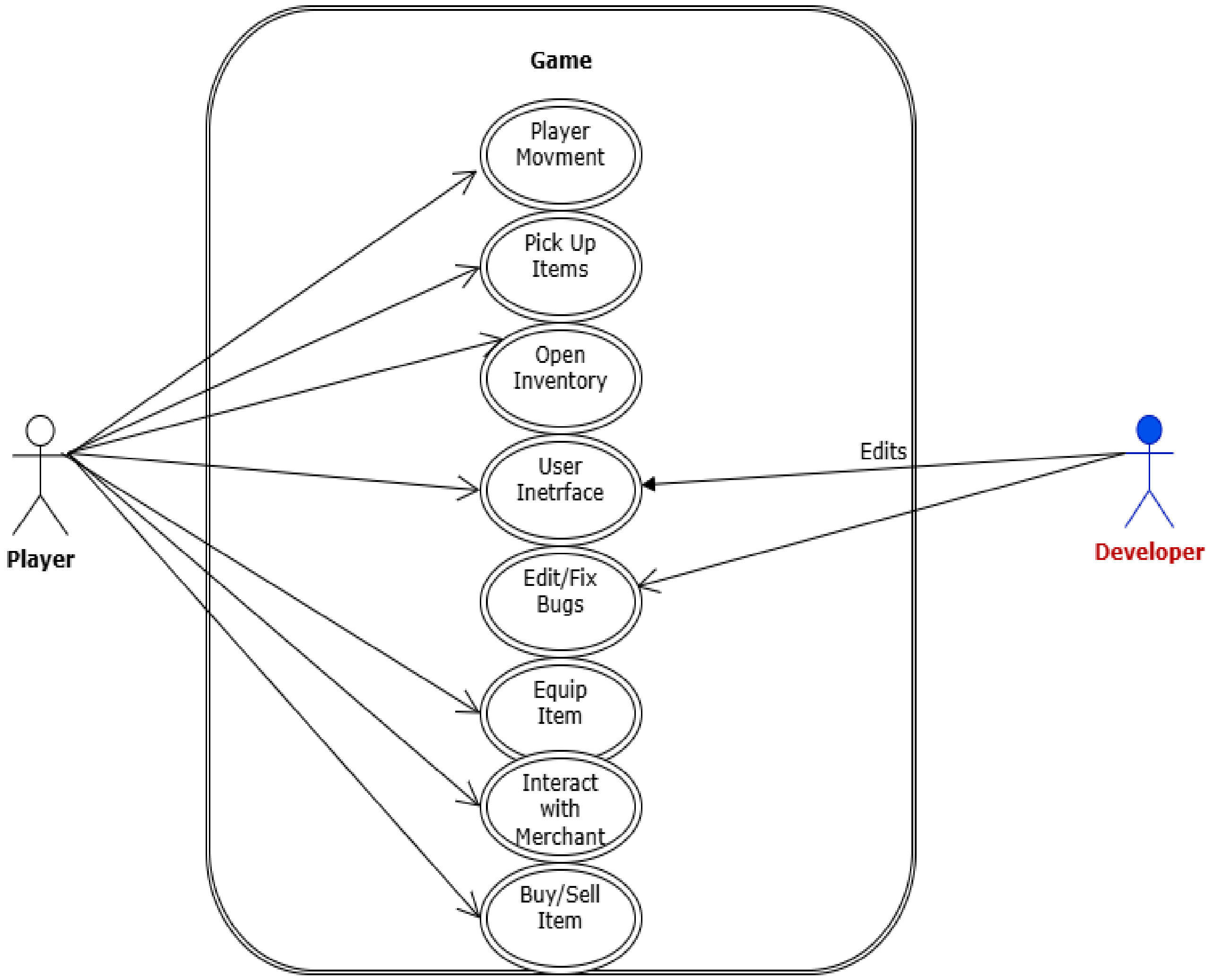
5.1.1 Functional Requirements:

i The player must be able to move around the game world using keyboard inputs
ii The player must be able to collect items from the environment.
iii The player must be able to open an inventory and equip/use items.
iv The player must be able to attack enemies using basic attacks.
v Enemies must be able to damage the player when collisions occur.
vi The player must be able to interact with merchants and purchase items.
vii Health systems must update based on combat interactions.

5.1.2 Non-Functional Requirements:

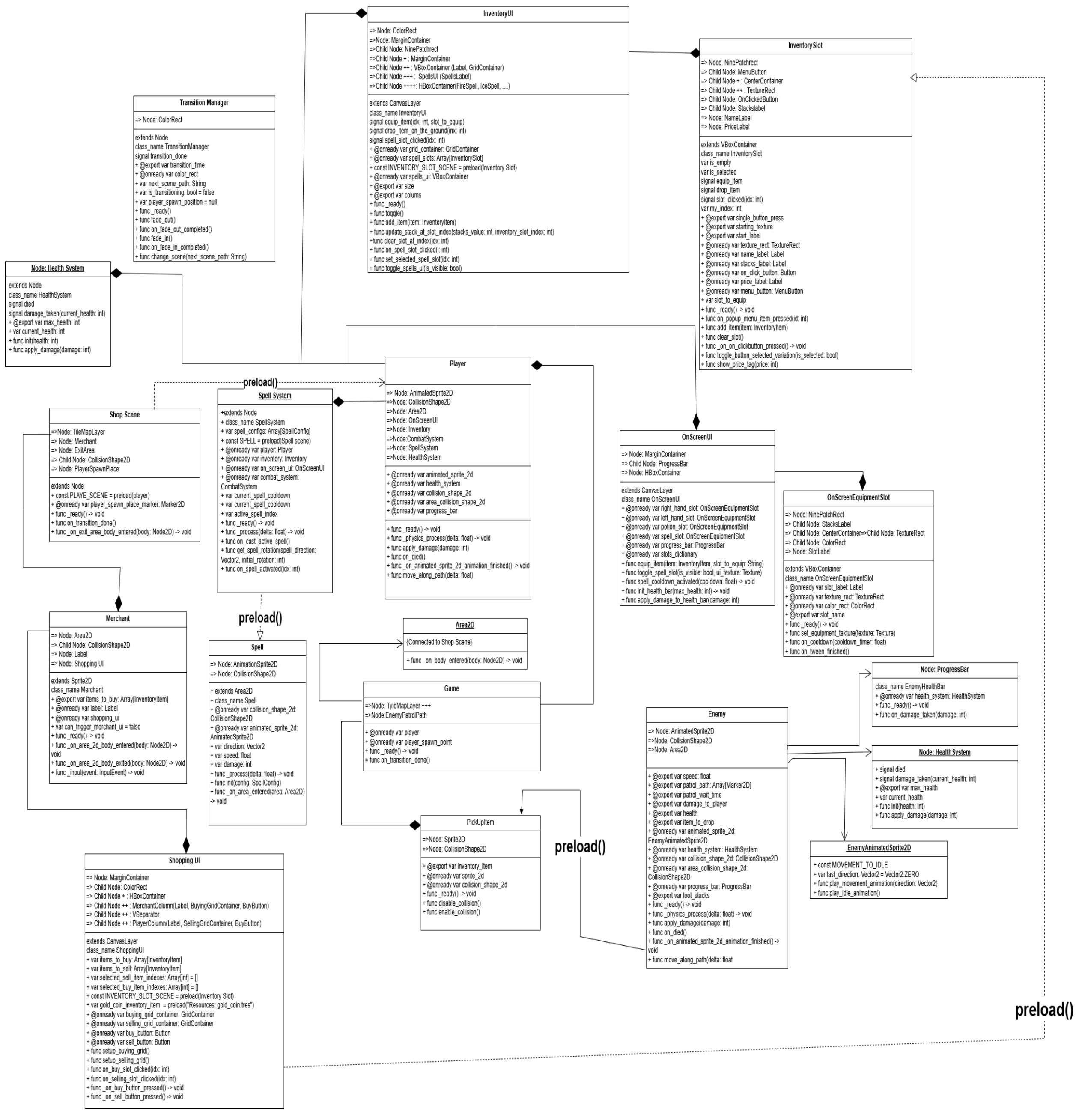
i The game must be playable on a standard Windows 11 machine with 8GB RAM.
ii The game must use free or open-source assets to avoid licensing issues.
iii The game should maintain a frame rate above 30 FPS on the target device.
iv Code must be version-controlled using GitHub.
v Basic error handling must be in place for key features (e.g., missing assets, invalid inputs).

5.2 Use Case Diagram



The use case diagram illustrates the core interactions between the player and the game system. These include fundamental actions such as moving the character, picking up items, equipping gear, and attacking enemies. It also includes system-level interactions like opening the inventory, using spells, and engaging with merchants. This high-level overview defines the scope of player interactions and guided the development of key gameplay mechanics during implementation.

5.3 Class and Scene Interaction Diagram



```
InventoryUI
=> Node: ColorRect
=>Node: MarginContainer
=>Child Node: NinePatchrect
=>Child Node + : MarginContainer
=>Child Node ++ : VBoxContainer (Label, GridContainer)
=>Child Node +++ :  SpellsUI (SpellsLabel)
=>Child Node ++++: HBoxContainer(FireSpell, IceSpell, ....)
extends CanvasLayer
class_name InventoryUI
signal equip_item(idx: int, slot_to_equip)
signal drop_item_on_the_ground(inx: int)
signal spell_slot_clicked(idx: int)
+ @onready var grid_container: GridContainer
+ @onready var spell_slots: Array[InventorySlot]
+ const INVENTORY_SLOT_SCENE = preload(Inventory Slot)
+ @onready var spells_ui: VBoxContainer
+ @export var size
+ @export var colums
+ func _ready()
+ func toggle()
+ func add_item(item: InventoryItem)
+ func update_stack_at_slot_index(stacks_value: int, inventory_slot_index: int)
+func clear_slot_at_index(idx: int)
+ func on_spell_slot_clicked(i: int)
+ func set_selected_spell_slot(idx: int)
+ func toggle_spells_ui(is_visible: bool)
```

```
InventorySlot
=> Node: NinePatchrect
=> Child Node: MenuButton
=> Child Node + : CenterContainer
=> Child Node ++ : TextureRect
=> Child Node: OnClickedButton
=> Child Node: Stackslabel
=> Node: NameLabel
=> Node: PriceLabel
extends VBoxContainer
class_name InventorySlot
var is_empty
var is_selected
signal equip_item
signal drop_item
signal slot_clicked(idx: int)
var my_index: int
+ @export var single_button_press
+ @export var starting_texture
+ @export var start_label
+ @onready var texture_rect: TextureRect
+ @onready var name_label: Label
+ @onready var stacks_label: Label
+ @onready var on_click_button: Button
+ @onready var price_label: Label
+ @onready var menu_button: MenuButton
+ var slot_to_equip
+ func _ready() -> void
+ func on_popup_menu_item_pressed(id: int)
+ func add_item(item: InventoryItem)
+ func clear_slot()
+ func _on_on_clickbutton_pressed() -> void
+ func toggle_button_selected_variation(is_selected: bool)
+ func show_price_tag(price: int)
```

```
Transition Manager
=> Node: ColorRect
extends Node
class_name TransitionManager
signal transition_done
+ @export var transition_time
+ @onready var color_rect
+ var next_scene_path: String
+ var is_transitioning: bool = false
+ var player_spawn_position = null
+ func _ready()
+ func fade_out()
+ func on_fade_out_completed()
+ func fade_in()
+ func on_fade_in_completed()
+ func change_scene(next_scene_path: String)
```

```
Node: Health System
extends Node
class_name HealthSystem
signal died
signal damage_taken(current_health: int)
+ @export var max_health: int
+ var current_health: int
+ func init(health: int)
+ func apply_damage(damage: int)
```

```
Shop Scene
=>Node: TileMapLayer
=> Node: Merchant
=> Node: ExitArea
=> Child Node: CollisionShape2D
=> Node: PlayerSpawnPlace
extends Node
+ const PLAYE_SCENE = preload(player)
+ @onready var player_spawn_place_marker: Marker2D
+ func _ready() -> void
+ func on_transition_done()
+ func _on_exit_area_body_entered(body: Node2D) -> void
```

```
Spell System
+extends Node
+ class_name SpellSystem
+ var spell_configs: Array[SpellConfig]
+ const SPELL = preload(Spell scene)
+ @onready var player: Player
+ @onready var inventory: Inventory
+ @onready var on_screen_ui: OnScreenUI
+ @onready var combat_system:
CombatSystem
+ var current_spell_cooldown
+ var current_spell_cooldown
+ var active_spell_index
+ func _ready() -> void
+ func _process(delta: float) -> void
+ func on_cast_active_spell()
+ func get_spell_rotation(spell_direction:
Vector2, initial_rotation: int)
+ func on_spell_activated(idx: int)
```

```
Player
=> Node: AnimatedSprite2D
=>Node: CollisionShape2D
=>Node: Area2D
=>Node: OnScreenUI
=>Node: Inventory
=>Node:CombatSystem
=>Node: SpellSystem
=>Node: HealthSystem
+ @onready var animated_sprite_2d
+ @onready var health_system
+ @onready var collision_shape_2d
+ @onready var area_collision_shape_2d
+ @onready var progress_bar
+ func _ready() -> void
+ func _physics_process(delta: float) -> void
+ func apply_damage(damage: int)
+ func on_died()
+ func _on_animated_sprite_2d_animation_finished() -> void
+ func move_along_path(delta: float)
```

```
OnScreenUI
=> Node: MarginContariner
=> Child Node: ProgressBar
=> Node: HBoxContainer
extends CanvasLayer
class_name OnScreenUI
+ @onready var right_hand_slot: OnScreenEquipmentSlot
+ @onready var left_hand_slot: OnScreenEquipmentSlot
+ @onready var potion_slot: OnScreenEquipmentSlot
+ @onready var spell_slot: OnScreenEquipmentSlot
+ @onready var progress_bar: ProgressBar
+ @onready var slots_dictionary
+ func equip_item(item: InventoryItem, slot_to_equip: String)
+ func toggle_spell_slot(is_visible: bool, ui_texture: Texture)
+ func spell_cooldown_activated(cooldown: float) -> void
+ func init_health_bar(max_health: int) -> void
+ func apply_damage_to_health_bar(damage: int)
```

```
OnScreenEquipmentSlot
=> Node: NinePatchRect
=> Child Node: StacksLabel
=> Child Node: CenterContainer=>Child Node: TextureRect
=> Child Node: ColorRect
=> Node: SlotLabel
extends VBoxContainer
class_name OnScreenEquipmentSlot
+ @onready var slot_label: Label
+ @onready var texture_rect: TextureRect
+ @onready var color_rect: ColorRect
+ @export var slot_name
+ func _ready() -> void
+ func set_equipment_texture(texture: Texture)
+ func on_cooldown(cooldown_timer: float)
+ func on_tween_finished()
```

```
Merchant
=> Node: Area2D
=> Child Node: CollisionShape2D
=> Node: Label
=> Node: Shopping UI
extends Sprite2D
class_name Merchant
+ @export var items_to_buy: Array[InventoryItem]
+ @onready var label: Label
+ @onready var shopping_ui
+ var can_trigger_merchant_ui = false
+ func _ready() -> void
+ func _on_area_2d_body_entered(body: Node2D) ->
void
+ func _on_area_2d_body_exited(body: Node2D) -> void
+ func _input(event: InputEvent) -> void
```

```
Spell
=> Node: AnimationSprite2D
=> Node: CollisionShape2D
+ extends Area2D
+ class_name Spell
+ @onready var collision_shape_2d:
CollisionShape2D
+ @onready var animated_sprite_2d:
AnimatedSprite2D
+ var direction: Vector2
+ var speed: float
+ var damage: int
+ func _process(delta: float) -> void
+ func init(config: SpellConfig)
+ func _on_area_entered(area: Area2D)
-> void
```

```
Area2D
{Connected to Shop Scene}
+ func _on_body_entered(body: Node2D) -> void
```

```
Game
=>Node: TyleMapLayer +++
=>Node:EnemyPatrolPath
+ @onready var player
+ @onready var player_spawn_point
+ func _ready() -> void
= func on_transition_done()
```

```
Node: ProgressBar
class_name EnemyHealthBar
+ @onready var health_system: HealthSystem
+ func _ready() -> void
+ func on_damage_taken(damage: int)
```

```
Enemy
=> Node: AnimatedSprite2D
=>Node: CollisionShape2D
=>Node: Area2D
+ @export var speed: float
+ @export var patrol_path: Array[Marker2D]
+ @export var patrol_wait_time
+ @export var damage_to_player
+ @export var health
+ @export var item_to_drop
+ @onready var animated_sprite_2d:
EnemyAnimatedSprite2D
+ @onready var health_system: HealthSystem
+ @onready var collision_shape_2d: CollisionShape2D
+ @onready var area_collision_shape_2d:
CollisionShape2D
+ @onready var progress_bar: ProgressBar
+ @export var loot_stacks
+ func _ready() -> void
+ func _physics_process(delta: float) -> void
+ func apply_damage(damage: int)
+ func on_died()
+ func _on_animated_sprite_2d_animation_finished() ->
void
+ func move_along_path(delta: float
```

```
Node: HealthSystem
+ signal died
+ signal damage_taken(current_health: int)
+ @export var max_health
+ var current_health
+ func init(health: int)
+ func apply_damage(damage: int)
```

```
EnemyAnimatedSprite2D
+ const MOVEMENT_TO_IDLE
+ var last_direction: Vector2 = Vector2.ZERO
+ func play_movement_animation(direction: Vector2)
+ func play_idle_animation()
```

```
PickUpItem
=>Node: Sprite2D
=>Node: CollisionShape2D
+ @export var inventory_item
+ @onready var sprite_2d
+ @onready var collision_shape_2d
+ func _ready() -> void
+ func disable_collision()
+ func enable_collision()
```

```
Shopping UI
=> Node: MarginContainer
=> Child Node: ColorRect
=> Child Node + : HBoxContainer
=> Child Node ++ : MerchantColumn(Label, BuyingGridContainer, BuyButton)
=> Child Node ++ : VSeparator
=> Child Node ++ : PlayerColumn(Label, SellingGridContainer, BuyButton)
extends CanvasLayer
class_name ShoppingUI
+ var items_to_buy: Array[InventoryItem]
+ var items_to_sell: Array[InventoryItem]
+ var selected_sell_item_indexes: Array[int] = []
+ var selected_buy_item_indexes: Array[int] = []
+ const INVENTORY_SLOT_SCENE = preload(Inventory Slot)
+ var gold_coin_inventory_item  = preload("Resources: gold_coin.tres")
+ @onready var buying_grid_container: GridContainer
+ @onready var selling_grid_container: GridContainer
+ @onready var buy_button: Button
+ @onready var sell_button: Button
+ func setup_buying_grid()
+ func setup_selling_grid()
+ func on_buy_slot_clicked(idx: int)
+ func on_selling_slot_clicked(idx: int)
+ func _on_buy_button_pressed() -> void
+ func _on_sell_button_pressed() -> void
```

The class and scene interaction diagram (see appendix) presents a detailed representation of how the game's core systems are structured and interlinked in Godot. It includes classes such as Player, InventoryUI, Enemy, SpellSystem, ShopScene, and CombatSystem, all annotated with their nodes, signals, variables, and methods. Because of its scale and complexity, the diagram is best viewed with zoom to explore each module's full detail.

The Player node acts as the central controller, interfacing with the health system, spell system, inventory backend, and multiple UI elements. Enemies use a separate HealthSystem instance and engage the Player through collision and damage methods. Signals like body_entered, custom UI events, and inventory slot clicks facilitate communication between scenes. UI components like InventoryUI, OnScreenUI, and ShoppingUI are separated into their own structured sub-scenes, maintaining a clean separation of logic and layout.

This modular structure, made possible by Godot's node-based architecture and scripting, was carefully designed to follow principles of low coupling and high cohesion. This approach aligns with software engineering best practices, enabling better maintainability, reusability, and clearer debugging paths throughout development. I will include the file within the documents folder of the project.

6. Professional Issues

Game development, particularly at the academic and indie levels, requires not only technical proficiency but a strong understanding of professional and ethical responsibilities. This section critically reflects on legal, ethical, accessibility, and practical considerations encountered during the development of *Wimbledon's Lot*, while also incorporating lessons learned from hardware limitations, resource constraints, and initial overconfidence at the project's outset.

6.1 Copyright and Licensing

From the beginning, the project was designed to respect the intellectual property rights of asset creators. All graphical, audio, and UI elements used in *Wimbledon's Lot* were sourced from open-source and royalty-free repositories, primarily Itch.io and OpenGameArt.org. These platforms provide resources under permissive licenses such as Creative Commons Attribution (CC-BY) or CC0, which allow for use in both commercial and non-commercial projects, often with minimal attribution requirements.

A critical professional responsibility was the verification of each asset's license terms. During development, extra care was taken to avoid including any resources with restrictive or non-commercial clauses, and appropriate attribution was noted in internal documentation. This is in line with ethical practices highlighted by Griffith, who emphasizes that asset sourcing must be transparent and compliant with the project's scope [19].

In terms of engine licensing, Godot's open-source MIT license is highly permissive and allows full freedom to customize and redistribute code. This aligns with the broader ethics of open-source development, which prioritizes collaboration, transparency, and freedom of use — principles that proved invaluable throughout this project.

6.2 Accessibility and Inclusive Design

Although the primary focus was on mechanics and functional completeness, accessibility remained an underlying consideration. UI scaling was implemented using MarginContainers and NinePatchRects in Godot to ensure compatibility across different screen resolutions. Control input was mapped to both keyboard and customizable bindings using Godot's InputEvent system [17].

Nonetheless, several limitations remain. Features such as remappable controls, screen reader support, and visual impairment accommodations were not implemented, primarily due to time constraints and technical unfamiliarity. In future versions, more inclusive design should be considered, including subtitles for audio, color-blind friendly palettes, and multiple input schemes.

As Redmond et al. argue in their review of real-time software accessibility, accessibility cannot be an afterthought — it must be part of the design ethos from the beginning [20]. While basic accessibility infrastructure was considered, further enhancements would be necessary to meet modern industry standards.

6.3 Development Ethics and Resource Management

One of the core ethical challenges was overpromising within the initial project proposal. The original plan envisioned a multi-level game, with advanced AI, dynamic quest trees, a fully functional merchant system, and online features like multiplayer or leaderboards. However, as development progressed, it became clear that these ambitions were not feasible within the available time and hardware resources.

The project was developed entirely on a single mid-range laptop with 8GB RAM and an Intel i7 processor — sufficient for basic development but limiting for testing more performance-heavy features like AI pathfinding or post-processing effects. Working without a second monitor or external peripherals added to the fatigue, especially during intensive debugging sessions or when managing multiple editor windows simultaneously.

In hindsight, these limitations mirrored larger issues in the industry. High-profile examples like *Cyberpunk 2077* serve as cautionary tales: overambition, scope mismanagement, and unrealistic deadlines can compromise quality and credibility. Just as CD Projekt had to scale back features and change their marketing claims post-launch, I too had to revise the scope of *Wimbledon's Lot* — dropping some planned systems in favor of producing a polished, well-tested first level [18].

This form of ethical self-correction — recognizing overreach and responsibly narrowing scope — is an essential part of software professionalism, even in solo or academic projects.

6.4 Plagiarism and Originality

An interesting professional issue arose around the question of plagiarism. In game development, where templates, tutorials, and shared codebases are common, it can be difficult to delineate originality from reuse. However, the use of tutorials or example code is ethically permissible when properly transformed and attributed.

In this project, all logic systems — including movement, combat, and inventory — were either self-authored or significantly modified from base templates, such as those found in Godot's official documentation [17]. Key code like the Player.gd movement system was refactored several times to meet the game's needs. Moreover, debugging complex interactions — like signals between InventorySlot and InventoryUI — highlighted the unique structure of the game's logic.

This aligns with Hadizadeh et al.'s perspective that originality in programming often lies in integration and problem-solving rather than starting from zero [21].

6.5 Self-Reflection and Psychological Pressure

Perhaps the most personal ethical issue was managing overconfidence and under-preparation. At the beginning, much time was spent on writing a detailed

Game Design Document (GDD), with the assumption that the technical aspects — like scripting player movement — would be easier than they turned out to be.

Debugging early movement code took days. Simple errors, like attaching a script to the wrong node or configuring physics bodies improperly, consumed time and caused considerable stress. These experiences were humbling and forced a recalibration of expectations.

The solo nature of the project also contributed to psychological strain. Working alone, often late into the night, created a sense of isolation and imposter syndrome. As noted in the diary: "This progress is highly anti-social… [it] could lead to physical health issues or psychological issues like imposter syndrome". This kind of reflection is vital. A responsible developer acknowledges not just technical failures, but also the emotional toll that solo creative work can take.

Balancing ethical labor — doing what's right for the product — with personal health became a central concern. Future projects should be scoped with not only feature lists, but *sustainability* of effort in mind.

6.6 Use of Version Control and Data Integrity

Using GitHub and Git Bash throughout development was not just a technical best practice, but an ethical one. Frequent commits, use of branches (e.g., player1.0), and command-line version restoration (git restore) protected against data loss and enabled safe experimentation.

This practice proved critical when debugging or reversing destructive changes, such as accidentally removing resource files or breaking script links. As emphasized by Schwalbe, version control is not just about history management — it's about accountability and integrity in development cycles [22].

7. Evaluation and Reflection & Next Steps

7.1 Evaluation and Reflection

The development of *Wimbledon's Lot* offered a deep and multifaceted learning experience, stretching beyond technical implementation into areas of project management, problem-solving, and personal resilience. When evaluated against the original project plan, the outcome reflects a clear case of adapting to real-world constraints and revising goals to focus on quality over scope.

From a technical standpoint, key objectives were successfully delivered: the game features a fully functional movement system, inventory backend, item pickup interactions, and a layered overworld environment. Systems such as health tracking, player UI, inventory UI, and merchant interaction were implemented and tested in a modular, maintainable manner. While the initial plan included more advanced features such as AI enemy pathfinding, puzzle logic, multiplayer integration, and leaderboard systems, these were ultimately set aside due to technical complexity, time limitations, and hardware constraints.

This process mirrors industry realities where scope often changes based on resources and feasibility. Rather than continuing to chase unmanageable goals, the decision to focus on polish, modularity, and stability proved far more beneficial in terms of learning and producing a usable prototype.

In terms of personal growth, the project revealed areas of overconfidence early on — particularly with underestimating the time needed to complete core systems like movement, animation syncing, and player-enemy interactions. Mistakes like spending too much time early on perfecting the Game Design Document, without first validating the technical pipeline, led to time loss and pressure toward the final stages. Debugging fundamental mechanics taught valuable lessons in patience, documentation, and testing workflows.

Additionally, emotional and psychological challenges were significant. The isolation of working solo for long periods, combined with setbacks like Git corruption, scene loss, and repeated refactoring, contributed to imposter syndrome and burnout. However, these struggles became opportunities for reflection. Learning how to pace work, simplify goals, and accept imperfection were among the most important takeaways.

From a professional standpoint, the experience highlighted the importance of documentation (via GitHub and diary entries), licensing compliance, proper asset sourcing, and ethical scoping. Working with open-source tools and assets provided insight into the indie development ecosystem and helped build foundational habits such as reading licenses, citing sources, and using documentation effectively.

7.2 Next Steps and Future Improvements

With a stable prototype established, the next phase of development for *Wimbledon's Lot* could explore several promising extensions:

Technical Enhancements:

1) **AI Pathfinding**: Implementing dynamic patrols and pursuit behaviours using A* algorithms or Godot's Navigation2D system.
2) **Save/Load System**: Adding persistent game states using JSON or Godot's File API to store inventory, player position, and stats.
3) **Combat System Expansion**: Introducing projectile spells, enemy variation, and player damage feedback animations.
4) **Quest System**: Creating modular quests with branching outcomes using finite state machines and dialogue condition checks.

Gameplay and World:

1) **Multiple Levels**: Building additional zones beyond Castle Rock, with gated progression based on inventory or quest completion.

2) **Puzzle Mechanics**: Integrating environmental puzzles (switches, movable blocks, etc.) to enhance level variety.
3) **Merchant Expansion**: Allowing item sell-back, inventory restocks, and price variation.

Accessibility and UX:

1) **Remappable Controls**: Implementing a keybind system to enhance accessibility.
2) **Accessibility Settings**: Adding visual assist tools (color filters, UI scaling, subtitles).
3) **UI Polishing**: Refining inventory slots, animations, and tooltips for smoother experience.

Professional Packaging:

1) **Bug Fix Logs**: Adding a changelog and structured bug tracker to prepare for alpha/beta release.
2) **Open Source Release**: Publishing on GitHub with documentation for contributions.
3) **Demo Video & Portfolio Upload**: Creating a clean demo presentation and hosting on itch.io, GitHub, or a personal portfolio site.

In short, the project lays a solid technical and conceptual foundation. With more time, resources, and collaboration, *Wimbledon's Lot* has the potential to grow into a fully realized narrative-driven RPG-lite experience.

8. Final Conclusion

The development of *Wimbledon's Lot* served as a comprehensive exploration into independent game design, blending creativity with software engineering principles. Over the course of the project, core gameplay mechanics were implemented using the Godot engine, supported by a structured inventory system, collision-driven interactions, and modular UI elements. Despite technical setbacks and time constraints, the prototype successfully demonstrates foundational elements of a 2D top-down adventure game.

This journey extended beyond code — it offered deep insights into planning, resilience, ethical practice, and personal growth. Starting with limited technical knowledge and no prior game development experience, the project matured into a working, extensible prototype through persistent effort and research. It highlights how theoretical frameworks such as Flow Theory, Environmental Storytelling, and Bartle's Taxonomy can shape not just gameplay, but also development strategy and player engagement.

Perhaps most importantly, the experience emphasized the balance between ambition and feasibility. Through reflective practice and ongoing problem-solving, the project transformed initial overreach into a clear, modular, and meaningful game environment. The lessons learned in scope management, documentation, and self-directed learning will directly inform future technical and professional endeavors.

9. Bibliography

[1. Csikszentmihalyi, M. (1990). *Flow: The Psychology of Optimal Experience*. New York: Harper & Row.

[2. Bartle, R. (1996). *Hearts, Clubs, Diamonds, Spades: Players Who Suit MUDs*. Journal of MUD Research. Available at: https://mud.co.uk/richard/hcds.htm [Accessed 27 Apr. 2025].

[3. Jenkins, H. (2004). *Game Design as Narrative Architecture*. In: Wardrip-Fruin, N. and Harrigan, P. (eds.) *First Person: New Media as Story, Performance, and Game*. MIT Press.

[4. Marchand, A. and Hennig-Thurau, T. (2013). *Value Creation in the Video Game Industry: Industry Economics, Consumer Benefits, and Research Opportunities*. *Journal of Interactive Marketing*, 27(3), pp.141–157.

[5. Hamari, J. and Järvinen, A. (2011). *Building Customer Relationship through Game Mechanics in Social Games*. In: *Business, Technological and Social Dimensions of Computer Games: Multidisciplinary Developments*. IGI Global.

[6. Gamma, E. et al. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[7. Freeman, E. et al. (2004). *Head First Design Patterns*. O'Reilly Media.

[8. Sommerville, I. (2016). *Software Engineering*. 10th ed. Pearson Education.

[9. Marchand, A. and Hennig-Thurau, T. (2013). [Duplicate of Ref. 4]

[10. Norman, D. A. (2013). *The Design of Everyday Things*. Revised and Expanded Edition. MIT Press.

[11. Schell, J. (2019). *The Art of Game Design: A Book of Lenses*. 3rd ed. CRC Press.

[12. Fullerton, T. (2014). *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*. 3rd ed. CRC Press.

[13. Godot Engine Community. (2023). *Godot Asset Library*. Available at: https://godotengine.org/asset-library [Accessed 25 Apr. 2025]

[14. OpenGameArt.org. (n.d.). *Free Game Assets*. Available at: https://opengameart.org [Accessed 24 Apr. 2025].

[15. Zechner, M. and Green, R. (2012). *Beginning Android Games*. 2nd ed. Apress.

[16. Nystrom, R. (2014). Game Programming Patterns. Genever Benning. Available at: http://gameprogrammingpatterns.com/ [Accessed 26 Apr. 2025].

[17. Godot Engine Documentation. (2024). *Godot Engine (stable branch) Official Documentation*. Available at: https://docs.godotengine.org/en/stable/index.html [Accessed 27 Apr. 2025].

[18. Chalk, A. (2020). *Cyberpunk 2077 cuts and missing features: What CD Projekt promised, and what happened*. *PC Gamer*. Available at: https://www.pcgamer.com/cyberpunk-2077-missing-features-cut-content/ [Accessed 28 Apr. 2025].

[19. Griffith, C. (2012). *Real-World Flash Game Development*. Routledge, 2nd Ed.

[20. Redmond, P., Jones, L., & Williams, H. (2024). *Exploring the Theory and Practice of Concurrency in ECS*. ACM Transactions on Interactive Intelligent Systems, 14(2), pp.105–118.

[21. Hadizadeh, M., Keller, B. and Tran, D. (2024). *Algorithm-Free Programming with a Bio-Inspired Computing Machine*. *Software: Practice and Experience*, 54(1), pp.1–25.

[22. Schwalbe, K. (2020). *Information Technology Project Management*. 9th ed. Cengage Learning.

10. Appendices

Appendix A – Development Diary

My Project diary

---Mohammad Ibrahim Salman---

created: 30/10/2024

Entry: 05/11/2024

For my initial stage of development, I will create a Game Design Document (GDD),

That will serve as a blueprint for the development team and helps ensure everyone involved has a clear understanding of the project.

My GDD will include: Game Overview, Story and Setting, Gameplay Machanics, Level Design, Characters and Enemies, and User Interface (UI).

I will be writting my GDD while simultaneously woring on the inital phase of the game, that is importing the assets.

And more importantly start preparing right away.

Having a idea of what the game will look like, I will proceed with creating nodes as this engine accommodates developere if there were a change in vision, like moving from a 2-D game to a 3-D one.

For this projet, I will be developing 2-D game for the sake of a first time developer.

As for the GDD, I will make a word document and store it in the Documents Folder with the Project Plan.

, while importing the gaming assets from open source websites.

I will make notes on new information I read, tasks I preformed, and where I will be needing to go.

Imported the assets.

I may seen like I have imported too much, though there are two reasons for that:

1) Some packages include a variety of textures, characters, and objects to use, thus there'll be more options for decorating the levels when constructing the Nodes and the Scenes{I'll explain later}.

2) If there are assets that I end up not using, I'll probably remove them.

The assets are from open source websites where are as follows:

1) https://itch.io/game-assets/free/tag-top-down

2) https://opengameart.org/

In Godot, a node is a fundamental building block with a specific function (e.g., Sprite, Audio, Script) that represents an element of game logic or behaviour.

A scene is a collection of nodes arranged in a tree hierarchy, allowing complex objects or environments to be created and reused throughout the game.

This structure promotes modularity and simplifies the development process

I resolved an issue with sound asset imports in Godot that were causing errors on Windows.

The problem stemmed from macOS-specific metadata files (__MACOSX folders and ._ files) embedded in the sound pack, which Windows couldn’t interpret correctly.

These extra files were purely metadata that macOS adds, and they aren't actually needed for the game.

I deleted the unnecessary files and re-imported the sound assets directly, clearing up the import errors.

It's a good reminder of the platform differences that can sneak into asset management!

Entry: 12/11/2024

Today, I delved into what a Game Design Document (GDD) truly is and the essential role it plays in game development.

Essentially, a GDD is a comprehensive blueprint for a game, detailing everything from the big-picture vision to the smallest mechanics, ensuring that all team members are aligned and that ideas are consistently documented.

The GDD's outline usually includes several sections:

a. Introduction, which sets the tone and goals of the project;
b. Game Overview, where the game concept and target audience are outlined;
c. Gameplay, explaining the core mechanics and controls;
d. Story and Narrative, describing the storyline and key themes;
e. Characters, detailing the main protagonists, antagonists, and any supporting NPCs;
f. Levels and Environments, breaking down each game location and its unique features;
g. User Interface (UI), outlining the menus and HUD elements;
h. Art and Visuals, discussing the style, colors, and overall aesthetic;
i. Audio, which covers the music, sound effects, and ambient sounds;
j. Technical Details, specifying the game engine, platform requirements, and technical constraints;

k. Marketing and Promotion, where target audiences and marketing strategies are planned;
l. Budget and Schedule, estimating costs and setting deadlines.

So far, I've drafted sections on the Introduction, Game Overview, Gameplay, Story and Narrative, Levels and Environments, and User Interface (UI).

Each of these sections is shaping up well, helping to solidify the core design and establish the game's atmosphere.

Yet, there's still more to cover, like Characters—which will bring depth to our protagonist and NPCs—and Art and Visuals, where I'll outline the game's stylistic direction.

The Audio section also needs fleshing out to set the right ambiance. Lastly, the Technical Details, Marketing and Promotion, and Budget and Schedule will round out the GDD, giving it the structure needed to take Wimbledon's Lot from concept to reality.

Entry: 13/11/2024

Today marks a milestone: I finally completed the remaining sections of the Game Design Document for Wimbledon's Lot.

Adding depth to the Characters section brought our protagonist and NPCs to life, giving each a unique story and purpose in the game world.

The Art and Visuals section now clearly defines the game's stylistic direction, inspired by vibrant, retro 32-bit graphics, ensuring a consistent visual atmosphere.

Audio is carefully outlined to set the perfect ambiance, from sound effects to atmospheric music.

I finished with the Technical Details, Marketing and Promotion, and Budget and Schedule sections, giving the GDD a robust, well-rounded structure.

Now, it’s a complete document, ready for others to read and envision Wimbledon’s Lot as a reality.

Entry: 17/11/2024

Created a new branch called player1.0 to focus on developing the main playable character.

Began by designing the protagonist that the player will control. First, I imported the character's idle animation, which consists of 7 frames, and increased its resolution for better visual quality.

Then, I set up a scene named Player (Idle) using a Node, where I incorporated the idle animation.

Additionally, I added an oval-shaped Collision Node to the scene to enable the character to interact with and collide with objects in the game world.

In the Game node, which includes player(idel), and the newly created camera in which is how the player will see the character they will be controlling.

Ran a successful animation.

Gnerating the script for the movement of the player, the engine generated the basic movement for the player as this:

```
extends CharacterBody2D


const SPEED = 300.0

const JUMP_VELOCITY = -400.0
```

```
func _physics_process(delta: float) -> void:
	# Add the gravity.
	if not is_on_floor():
		velocity += get_gravity() * delta

	# Handle jump.
	if Input.is_action_just_pressed("ui_accept") and is_on_floor():
		velocity.y = JUMP_VELOCITY

	# Get the input direction and handle the movement/deceleration.
	# As good practice, you should replace UI actions with custom gameplay
actions.
	var direction := Input.get_axis("ui_left", "ui_right")
	if direction:
		velocity.x = direction * SPEED
	else:
		velocity.x = move_toward(velocity.x, 0, SPEED)

	move_and_slide()
```

However, I changed the script to accomodated for the vertial movement of the player as for my intentions of the game I invisioned.

My code:

```
extends CharacterBody2D


const SPEED = 300.0

const JUMP_VELOCITY = -400.0


func _physics_process(delta: float) -> void:
	# Add gravity
	if not is_on_floor():
		velocity += get_gravity() * delta


	# Handle jump
	if Input.is_action_just_pressed("ui_accept") and is_on_floor():
		velocity.y = JUMP_VELOCITY


	# Get the input direction for both horizontal and vertical movement
	var input_vector = Vector2(
		Input.get_axis("ui_left", "ui_right"),
		Input.get_axis("ui_up", "ui_down")
	)


	# Normalize the direction to avoid diagonal speed being faster
	if input_vector != Vector2.ZERO:
		input_vector = input_vector.normalized()
```

```
    # Set the velocity based on the input
    velocity.x = input_vector.x * SPEED
    velocity.y += input_vector.y * SPEED  # If adding vertical movement directly


    # Move and slide with the updated velocity
    move_and_slide()
```

My first bug: When I run the program the character should move and jump, yet it would not respond to any keyboard input.

Fixed the bug:

Realized the issue was caused by mistakenly adding another CharacterBody2D node inside the existing Player node.

This redundancy confused the script as it couldn't determine which node to control.

After removing the extra CharacterBody2D node and ensuring the script was properly attached to the correct Player node, the character movement now works as expected. A good reminder to double-check the node structure before troubleshooting!

Updated the character image to a new set that includes animation frames for both horizontal and vertical movements, replacing the previous set that only supported horizontal motion.

This change allows for smoother and more dynamic animations, making the character's movement feel more natural and responsive in all directions.

Reorganized my assets into clearly labeled and easily identifiable folders to improve efficiency.

This new structure eliminates the need to shuffle through multiple folders to locate the required assets, streamlining the workflow and saving time during development.

I need to work on the following:

- Create animations for player movement in all directions.
- Determine if additional assets are needed for weapon usage and death scenes.
- Evaluate whether these assets are freely available or behind a paywall due to complexity.
- Address any bugs encountered during the process.
- Focus branch player1.0 on idle and run animations for player movement only.

Entry: 26/11/2024

I worked extensively on fixing and improving the player movement in my game.

Initially, I encountered several issues where the player character wouldn't move despite no errors or warnings in the script.

After detailed debugging, I realized the importance of properly setting up the node hierarchy and physics properties.

Specifically, I adjusted the CollisionShape2D and StaticBody2D nodes, ensuring they had the correct configurations and physics layers/masks to interact correctly.

I also rewrote the player.gd script to handle Zelda-style movement, adding logic to normalize input vectors for smooth diagonal movement and ensuring the move_and_slide() function worked as intended.

I clarified my understanding of input mapping in the project settings and confirmed the use of ui_left, ui_right, ui_up, and ui_down for movement.

Through this process, I learned to integrate animations properly by adding AnimationPlayer nodes and connecting them to player movement, ensuring seamless transitions between states like idle and running.

Lastly, I organized the code in the existing player.gd script rather than creating a new one, maintaining clarity and avoiding duplication.

Still trying to figure out the movement of the player and being able to controll the character.

All animations of idle, run, and attack are made facing all directions (death animantion will be implimented when creating enemies).

As off now, I'm getting the controlls working.

Got the controlls working. However I was able to perfom the basic movement of the idle animation.

The run animation and the attack (which I will soon impliment wll come about later), is yet to be implimented.

Also, need to get rid of a bug where the collision shape 2D sprite is still on the character while moving.

Fixed the bug where the collision shape 2D sprite is still on the character, by turning off the visibility under the Inspector tab.

Entry: 03/12/2024

Going to finish the character movement controlls.

Successfully implemented animation switching for my character in Godot!

Here’s a summary of how I achieved it:

To make the character play "Run" animations when moving (using WASD inputs) and transition to "Idle" animations when no input is provided,

while retaining the last movement direction.

Node Setup:

Added an AnimatedSprite2D node as a child of my Player node.

Loaded the necessary animations into the AnimatedSprite2D, including:

Idle animations: Idle_left, Idle_right, Idle_up, Idle_down.

Running animations: Run_left, Run_right, Run_up, Run_down.

Script Setup:

Attached a script to the Player node.

Declared constants and variables:

SPEED to control movement speed.

animated_sprite to reference the AnimatedSprite2D.

last_direction to track the last direction of movement for idle animations.

Movement Logic:

Used Input.get_action_strength to detect WASD inputs and calculate the movement vector.

Normalized the vector to ensure consistent speed in all directions.

Used move_and_slide() to move the character.

Animation Switching:

Created a get_direction function to determine the character's direction (left, right, up, or down) based on input.

Implemented a handle_animation function:

Played the "Run" animation when the character was moving.

Played the appropriate "Idle" animation when the character stopped, using last_direction to determine which idle animation to show.

Debugging and Fixing Indentation Issues:

Encountered a mixed tab/space indentation error.

Resolved it by converting all indentation to tabs for consistency.

Outcome:

The character now:

Transitions smoothly between "Run" animations while moving in any direction.

Remains in the correct "Idle" animation when no input is detected.

Retains the last movement direction for idle animations

Now to start creating the world.

There are many places and levels to create, but for now I'm going to start with the mai hub, where you first interact with the game.

I will be adding more assets for the world building and level design.

As of now, I am setting up the terrain.

16/02/2025

Today was a big day for my game development project!

I finally managed to wrap my head around creating tile sets and paths in Godot Engine.

It was a bit of a journey, but I'm thrilled with the progress.

First things first, I created a new scene and added a `TileMap` node, which is the foundation for building the game world.

The magic really happens in the Inspector tab. I started by selecting a `TileSet`.

This is where I define all the individual tiles that make up my terrain.

Creating the `TileSet` involved a few key steps.

I added "matching corners and edges," which is crucial for smooth transitions between different terrain types.

This prevents jarring visual glitches and makes the world look much more polished. I then added a new terrain type, which I cleverly named "Grass."

This is where I'll define how grass tiles behave and look.

The real fun began when I imported my "overworld" file.

This file contains a huge collection of tiles – not just grass, but also buildings, water, and all sorts of decorative elements that will bring my game world to life.

It’s like a treasure trove of artistic assets!

Next, I selected the "Grass" terrain type within my `TileSet`.

I then switched to the "Paint" tool and, under the "Terrains" section, I could actually *paint* the terrain onto my `TileMap`.

I chose my desired grass color and started laying down the foundations of my world.

It felt incredibly satisfying to see the landscape taking shape.

Now, for the tricky part: creating paths.

I'm still figuring out the most elegant way to handle this, but for now, I'm manually drawing the paths using the tile painting tools.

I’m thinking about using a separate tile layer for the path so that I can easily edit it later without affecting the surrounding terrain. I might even explore using Godot's navigation system for more dynamic pathfinding, but that's a challenge for another day.

The process of creating the path was quite manual.

I selected the appropriate path tiles from my overworld file and carefully painted them onto the tile map, trying to create smooth curves and natural-looking transitions.

It took some time and patience, but the result is starting to look pretty good.

The beauty of Godot's tile system is its flexibility.

I can easily add more terrain types, customize their properties, and paint them onto the map.

I can also create different layers for things like foreground elements, background details, and even interactive objects. This layering system is essential for creating complex and visually rich game worlds.

I’m excited to continue experimenting with tile sets and paths.

I want to explore different techniques for creating more dynamic and interesting level designs.

Perhaps I'll even look into procedural generation to create vast and varied landscapes automatically.

The possibilities seem endless!

23/02/2025

Today, I focused on populating my overworld with the basic building blocks of its environment:

the regular grass tiles. After setting up the `TileSet` and terrain types yesterday,

it was time to put those tools to practical use.

I opened my `TileMap` scene and made sure I had the "Grass" terrain selected in the Inspector.

Then, with the "Paint" tool active, I began laying down the foundation of the landscape.

It was a simple process, but it felt incredibly satisfying to see the world start to fill out.

I paid attention to creating a natural-looking distribution of grass, avoiding overly uniform patterns.

I varied the density and placement of the tiles to give the terrain a more organic feel.

I also made sure to leave space for other elements like paths, buildings, and water features,

which I'll be adding later.

The process was quite therapeutic, almost like digital gardening. I could see the overworld

taking shape, and it gave me a better sense of the scale and layout of the game.

I also realized the importance of planning the terrain layout early on, as it will

influence the placement of other game elements.

I'm getting more comfortable with Godot's tile system, and I'm looking forward to adding

more variety to the landscape. I'm thinking about incorporating different shades of grass,

as well as some subtle variations in tile placement to create a more visually

interesting environment.

I also ventured into adding water tiles to my overworld, bringing a new dimension to the landscape.

It was exciting to see how the water features transformed the overall look and feel of the map.

First, I imported the water tiles into my `TileSet` and defined a new terrain type for them.

I experimented with different water tile variations to find the right aesthetic,

settling on a combination of deep water and shallow water tiles.

Then, I carefully painted the water tiles onto the `TileMap`, creating lakes, rivers, and ponds.

I tried to create natural-looking shorelines and varied the depth of the water to give it a sense of realism.

I made sure to leave enough space for bridges and other decorations that I plan to add later.

Adding the water tiles made the overworld feel much more dynamic and interesting.

The reflections and subtle animations of the water added a layer of visual depth that was previously missing.

I also started planning the placement of bridges and other decorations.

I want to create a world that feels both natural and functional,

with clear paths and points of interest. I'm thinking about adding things like:

* Wooden bridges spanning rivers and lakes.

* Small docks and piers along the shorelines.

* Rocks and vegetation along the water's edge.

* Maybe even some floating objects or creatures in the water itself.

I'm excited to see how these decorations will further enhance the overworld.

I'm also thinking about adding interactive elements to the water,

such as fishing spots or hidden underwater areas.

The process of adding water tiles and planning decorations has been a lot of fun.

It's rewarding to see the overworld come to life, tile by tile.

I'm looking forward to continuing to build and refine the world,

adding more details and interactive elements as I go.

25/02/2025

Improved the water tiles to create rivers, by adding more tiles to the already created water with grass tileset.

26/02/2025

Today was all about expanding the overworld and laying the groundwork for key locations.

I spent a good chunk of time painting tiles to create a much larger landscape.

This included adding cliffs along certain edges, which instantly added verticality and visual interest.

I also mapped out areas where bridges would connect different sections of the map, ensuring smooth player traversal.

I'm visualizing wooden bridges spanning narrow ravines and stone bridges crossing wider rivers.

Most importantly, I began to define the area where the player's house will be located.

It'll serve as a central hub, potentially offering save points, a merchant, or even the starting point of the adventure.

I'm focusing on making this area visually distinct, perhaps with a small clearing or a unique terrain feature.

The sheer scale of the landscape is becoming apparent, and it's exciting to imagine the player exploring every corner.

I'm carefully considering the placement of each tile, ensuring the world feels both natural and purposeful.

Today,I tackled the challenge of creating bridge tiles that seamlessly integrate with the overworld, without clashing with the existing grassland tiles.

It was a meticulous process, but the results are definitely worth the effort.

The key was to create a separate `TileMap` node specifically for decorations, which I named "TileMap_Decorations."

This allowed me to paint bridge tiles (and other decorative elements) on top of the base grassland layer without causing any overlap or visual glitches.

It's like having two separate canvases stacked on top of each other, where I can paint on the top layer without affecting the bottom one.

In Godot, this is achieved by using multiple `TileMap` nodes and managing their Z-index (their stacking order).

By placing the "TileMap_Decorations" node above the main grassland `TileMap` node, I ensured that the bridge tiles would always appear on top.

Painting the bridge tiles was a delicate process.

I had to carefully align them with the edges of the grassland tiles, ensuring a smooth and natural transition.

It was like painting on a real canvas, where every stroke mattered. My mouse became my paintbrush, and the `TileMap` was my canvas, and I meticulously placed each tile.

It was a bit tedious, I'll admit.

But I know this attention to detail is crucial for creating a polished and immersive game world.

I'm willing to put in the grind to make sure everything looks just right.

The ability to layer `TileMap` nodes in Godot is incredibly powerful.

It allows for a great deal of flexibility and control over the level design.

I can now add all sorts of decorations, from bridges and fences to rocks and trees, without worrying about them interfering with the base terrain.

I'm starting to see the overworld come to life, and it's incredibly rewarding.

Each tile I place brings me one step closer to creating a playable game.

I'm excited to see how it all comes together in the end.

01/03/2025

Today, I embarked on the challenging task of adding cliff tiles to my overworld, transforming the landscape with dramatic elevation changes.

The vision was to have the main hub situated atop a cliff, with multiple cliff mountings allowing the player to ascend and descend.

It was a grand idea, but the execution proved to be quite the undertaking.

Constructing the cliffs was far more complex than I anticipated.

The overlapping of tile sets caused numerous headaches.

I found myself constantly deleting and erasing tiles, trying to achieve a clean and seamless look.

Managing the different layers of cliff tiles was particularly difficult.

There were instances where the grass tiles wouldn't align properly with the cliff edges, requiring me to painstakingly erase and redraw them. It felt like I was constantly battling the tiles, trying to force them into place.

At one point, I had to completely start over with the cliff construction.

The tiles were simply refusing to cooperate, and the result was a messy and disjointed landscape.

It was frustrating, but I knew I couldn't compromise on the quality of the terrain.

So, I took a deep breath and began again, carefully placing each tile and meticulously adjusting the layers.

After hours of painstaking work, I finally managed to create the cliffs I envisioned.

The main hub now sits proudly atop a towering cliff, with multiple pathways leading up and down.

It's a significant improvement, and I'm proud of the effort I put in.

Now, I'm moving on to the next stage: adding waterfalls, more cliff variations, and, most importantly, collision.

The collision is crucial for defining the boundaries of the world and preventing the player from walking through solid objects.

I'll need to create invisible walls along the cliff edges and around other obstacles.

I also plan to add more decorations to the main hub, making it feel more lived-in and inviting.

And, of course, I need to ensure that the character's movement is smooth and responsive.

The collision and movement mechanics are fundamental to the gameplay experience, and I want to get them right.

Today's work was a testament to the challenges of game development.

It's not always glamorous, and there are times when you have to grind through tedious tasks.

But the satisfaction of seeing your vision come to life makes it all worthwhile.

Following the work on the cliff terrain and hub area,

I decided to test the scene in-game.

That's when I encountered a rather glaring issue: the player was completely invisible!

After a bit of digging, I realized the problem stemmed from the scene hierarchy.

The player node, created early in development, was hidden behind the tilemap layers, effectively rendering it invisible.

To rectify this, I opted for a cleaner approach.

I removed the old player node and instantiated the player scene as a direct child of the main scene.

This simple change brought the player into view, correctly positioned above the tilemap. Finally, I could see my character moving within the hub!

However, this fix introduced a new challenge.

While the player was now visible and responsive, the camera remained stubbornly static. This was a critical oversight.

A static camera would break the player's immersion and make exploration cumbersome.

My next task was clear: implement a camera follow script.

The goal is to create a smooth, responsive camera that tracks the player's movements seamlessly.

Furthermore, the camera must respect the collision boundaries I've established.

It should stop moving when the player encounters an invisible wall, preventing the player from seeing beyond the playable area. Revealing the unfinished edges of the world would ruin the illusion of a complete and immersive environment.

This transition from terrain construction to camera implementation highlights the iterative nature of game development.

Each step forward reveals new challenges, requiring constant problem-solving and refinement. Now, I'm diving into scripting the camera follow, aiming for a smooth and polished player experience.

06/03/2025

Dear Diary,

I continued to refine the overworld, focusing on adding waterfalls and further sculpting the cliff terrain.

The core concept driving my design is creating a cohesive and contained hub world.

I want to establish clear boundaries, preventing the player from venturing beyond the intended playable area.

This is a common practice in 2D game design, especially in games that emphasize exploration within a defined space.

Think of it like creating a diorama or a miniature world.

You carefully construct the scenery within a box, and you want t empty space beyond. In my case,

the cliffshe viewer to appreciate the details within that box, not the and invisible walls serve as the "box," defining the limits of the player's exploration.

I'm being meticulous in placing each tile, ensuring that the water flows naturally, the cliffs rise convincingly,

and the transitions between different terrain types are seamless. It's akin to painting on a canvas, where every brushstroke contributes to the overall composition.

I'm focusing on the minute details, making sure that each water tile fits perfectly, and that the cliff edges are sharp and defined.

The idea is to create a sense of immersion, where the player feels like they're exploring a real, tangible place.

The boundaries, though invisible, are meant to be felt. The player should intuitively understand where they can and cannot go,

without encountering jarring transitions or empty spaces.

I've also been considering the placement of future decorative elements. Rocks, lily pads, and other details will add depth and visual interest to the water features.

These elements will further enhance the sense of realism and create a more engaging environment.

Collision is another crucial aspect of this process. I need to define the physical boundaries of the world,

preventing the player from walking through walls or falling off cliffs.

This will ensure that the player's movement feels grounded and responsive.

Finally, I'm looking forward to adding the houses and NPCs that will populate the hub world. These elements will bring the world to life,

adding narrative and gameplay depth.

I'm almost finished with the terrain itself, and I'm excited to move on to these more interactive and narrative-driven aspects of the game.

I'm pushing my skills to the limit, striving to create a polished and immersive game world.

The process is demanding, but the satisfaction of seeing the world take shape is incredibly rewarding.

A significant milestone today! I've finally completed the base terrain of the main hub world.

Every cliff, every waterfall, and every stretch of grassland is now in place.

All that remains is to breathe life into the scene through decorations.

I'm starting with the water features, adding rocks and lily pads to break up the monotony

and create a more natural look. After that, I'll scatter rocks across the land, adding

further visual interest. These simple decorations will go a long way in making the world

feel more lived-in.

With the terrain complete, I can now focus on the crucial task of implementing collision,

ensuring the player interacts with the world as intended.

I then embarked on the exciting task of decorating the hub world, bringing it to life with trees, rocks, and various foliage.

The goal is to create a visually rich and immersive environment, filled with natural details.

For smaller, repeating elements like bushes and fences, I'm continuing to use the tile painting method.

This allows for precise placement and seamless integration with the existing terrain.

I've created dedicated tile sets for these decorations, ensuring they blend harmoniously with the overall aesthetic.

However, for larger, more distinct objects like trees, rocks, and logs, I'm taking a different approach.

I'm utilizing the drag-and-drop functionality of the tile panel.

This allows me to place these objects directly into the overworld, arranging them in natural patterns and clusters.

It's like sculpting the environment, placing each object with care and consideration.

The key to this process is understanding the layering system within Godot.

I'm placing these decorations on the "TileMap_Decorations" layer, which sits above the base terrain layer.

This ensures that the objects appear on top of the ground tiles, preventing any visual overlap or glitches.

If I were to place these objects on the same layer as the terrain, they would be subject to the tile grid's limitations,

resulting in unnatural and potentially overlapping arrangements.

This layering technique provides a great deal of flexibility and control.

I can easily rearrange or remove decorations without affecting the underlying terrain.

It also allows me to create depth and visual variety, adding a sense of realism to the environment.

In the future, I plan to add even more decorative elements, further enhancing the overworld's visual appeal.

I'm excited to see how these details contribute to the overall atmosphere and immersion of the game.

11/03/2025

A quick but important fix today.

I noticed some of the tile map patterns, especially when viewed up close, appeared blurry.

It's a small detail, but it significantly impacts the overall visual quality.

I'll need to revisit the tile set textures and ensure they're optimized for the intended resolution.

Sharp, crisp tiles are essential for a polished look.

Seems like altering the Rendering Quandrant Size under the inspection tab fixed this issue.

The hub world is slowly transforming into a vibrant tapestry of nature.

Trees stand tall, huts nestled amongst the foliage, and flowers bloom in colorful clusters. Stones dot the landscape, and lily pads float serenely on the water's surface.

Each element, carefully placed, adds another layer of depth and charm. I could lose myself in this decorative dance, endlessly arranging and refining.

But the time has come to shift gears, to delve into the core mechanics that will breathe true life into this world.

The collision system now beckons, a crucial step towards a playable game.

The decorative phase, while fulfilling, is merely a prelude.

I must now build the invisible walls, the tangible boundaries that will define the player's experience.

It's a daunting task, but one I must embrace.

There's a nagging thought, though, a whisper of unfinished business. The waterfalls, fountains, and even the humble puddles of water – they yearn for movement, for animation.

Flags should flutter in the breeze, fountains should dance with water, and puddles should ripple with life.

This is a challenge for another time, a layer of polish that will add a touch of magic to the world. But for now, collision takes precedence.

The pressure is mounting. The final report looms, a testament to my dedication and progress.

I must pull myself together, channel my focus, and transform this beautiful canvas into a functional, engaging game.

It's a race against time, a test of my abilities, but I'm determined to see this through. The hub world, my creation, will come to life.

A wave of pure, unadulterated frustration washed over me today, threatening to drown my entire project.

I swear, sometimes this game development process feels like a cruel, twisted joke.

I was on the verge of progress, finally diving into the collision implementation on a fresh, clean branch. But fate, it seems, had other plans.

As soon as I switched to the new branch, a horrifying sight met my eyes: devastation.

My meticulously crafted decorations, the trees, the bushes, the houses – all reduced to gaping, maddening question marks.

It was as if a digital vandal had swept through my world, erasing everything I'd painstakingly placed. A pit of dread opened in my stomach. How? How could this have happened? I combed through the Git history, searching for clues, but found nothing. Just a gaping void where my decorations once stood.

I felt a surge of anger, a burning resentment towards the fickle nature of technology. Hours of work, vanished in an instant.

The meticulous placement, the careful consideration of each element – all for naught. It was a gut punch, a brutal reminder of the fragility of digital creations.

Now, I'm forced to backtrack, to return to the main branch and salvage what I can. I'll painstakingly recreate the decorations, ensuring every tree and bush is in its rightful place.

I'll commit and push, securing my work against the digital gremlins that seem determined to sabotage my efforts. And then, only then, will I dare to venture back into the collision branch.

It's a tedious, infuriating setback. But I refuse to be defeated. I'll rebuild, I'll persevere, and I'll emerge from this ordeal stronger.

The collision implementation will be done, the game will be built, and I'll prove that even the most frustrating glitches can't extinguish my determination.

The decorations are going back, I will get this right. One. Tile. At. A. Time.

"IMPORTNAT NOTE"

A cold dread settled over me today, a chilling mix of terror and frustration that I wouldn't wish on my worst enemy.

I attempted the seemingly simple task of merging my beautifully decorated world-building-1.0 branch into main. A task that should have been a triumphant step forward, instead spiraled into a nightmare.

The decorations, the carefully placed trees, rocks, and huts, simply refused to transfer. It was as if they were ghosts, fading into the digital ether.

I tried again, and again, tweaking, re-merging, but the results were the same. A sense of panic began to rise.

Then, the unthinkable happened. I opened Godot, and my game scene was gone.

Not just the decorations, but the entire scene, replaced by a stark, terrifying error message. It was a digital death knell. My heart pounded in my chest. Had I lost everything? Was my game, my creation, truly gone?

The error messages were a cryptic language of corrupted files and syntax errors, a chaotic jumble of technical jargon that seemed to mock my efforts.

I felt a wave of despair, a sense of utter helplessness. I turned to the digital oracle, ChatGPT, seeking answers, seeking salvation. It suggested I delve into the raw text of the game.tscn file, a daunting task for a weary developer.

But before I plunged into the abyss of code, I remembered the output debugger.

I opened it, my eyes scanning the lines, searching for the source of the catastrophe. It was a tedious process, a hunt for a needle in a digital haystack. And then, I found it. A series of misplaced equal signs, a missing bracket, tiny errors that had brought my world to its knees.

I opened the game.tscn file in Visual Studio Code, the raw, unedited heart of my scene.

The file, a complex structure of nodes and properties, was a testament to the intricate workings of Godot.

Each node, each property, carefully defined, held the key to my game. I painstakingly corrected the errors, carefully placing each bracket, each symbol, hoping against hope that I was restoring order to the chaos.

And then, miraculously, the scene loaded. But the victory was bittersweet. The decorations, the fruits of my labor, were gone.

Only the bare bones of the terrain remained. The TileMap_Decorations layer, the canvas upon which I had painted my world, was empty.

I felt a wave of exhaustion, a sense of having climbed a mountain only to find myself at the bottom again.

Collision implementation, the task that had seemed so daunting before, now felt like a distant dream.

I'm creating a new branch, decorate-world-building-decorations, and I will re-decorate.

I will add every tree, every rock, every hut, every single detail that was lost. I will rebuild my world, tile by painstaking tile.

The report deadline looms, a constant reminder of the pressure I'm under. But I will not be deterred.

I will not give up. I will rebuild, I will persevere, and I will create a game that I can be proud of.

This setback, though devastating, will not define me. I will rise from the ashes, stronger and more determined than ever.

As I began the painstaking process of redecorating, I noticed something peculiar.

When I went to the TileMap_Decorations node and examined its TileSet in the Inspector, the tileset.tres file was indeed present.

However, the tiles representing the trees and huts, sourced from separate tile sets, were still displaying those infuriating question marks.

It was a cruel twist, a reminder that the gremlins were still lurking.

After some thought, I decided against creating yet another branch for decorations.

The constant branch switching was clearly contributing to the chaos.

Instead, I'll commit to the main branch for all decoration-related work.

This simplifies the workflow and hopefully minimizes the risk of further mishaps.

Once the decorations are fully restored, I'll create a dedicated branch for collision implementation, keeping those critical mechanics isolated and secure.

19/03/2025

Today was a bit of a wrestle, but I think I've finally got the camera sorted (mostly).

I spent a good chunk of time trying to get it to follow the player smoothly in my top-down overworld.

Initially, it was just… stuck. Turns out, a simple zoom issue was throwing me off completely.

Who knew such a small thing could cause so much frustration?

After fiddling with the Camera2D node's properties, and a lot of double-checking my player's movement script, I managed to get it working.

The player moves, and the camera follows, which is exactly what I wanted. It's a relief to have that hurdle cleared.

Though, I'm still not entirely convinced it's perfect. I might need to tweak the smoothness and maybe add some subtle camera shake later on, just to give it a bit more life.

But for now, it's functional, and that's a win.

With the camera sorted, I'm finally moving on to fleshing out the rest of the overworld.

It's starting to feel more like a real game now. I've got a basic layout, but it's still pretty bare.

I need to add some more detailed scenery, maybe some NPCs wandering around, and definitely some points of interest.

This is where things are going to get more complex, and probably more time-consuming.

I'm also starting to think about the next steps beyond the basic layout.

Adding collision detection is going to be crucial, so players can't just walk through walls and objects.

And then there's the whole system for picking up items, managing an inventory, and interacting with the environment.

That's going to be a big undertaking, but I'm excited to dive into it.

I'm already brainstorming ideas for how to structure the inventory and how to make the item interactions feel intuitive.

I can feel the scope of this project expanding, and it's both exciting and a little daunting.

There's so much to do, but I'm determined to see it through. Each step, even the frustrating ones, feels like progress.

I'm learning so much, and I can't wait to see how this overworld evolves.

Alright, so things got a lot more… tangible today.

I spent a good portion of the afternoon tackling collision detection, and I've made some significant progress, but not without a few hiccups.

The main goal was to prevent the player from walking through the water and to keep them within the bounds of the main hub.

I'm happy to report that the water collision is working! I added collision shapes to the water tiles, and now the player is properly blocked.

It's a huge step towards making the overworld feel more solid and interactive.

However, I immediately ran into a rather annoying issue with the bridges.

It seems that the border tiles of the bridges are overlapping with the water collision shapes.

So, even though the bridges are visually above the water, the player is still being blocked as if they were trying to walk through the water itself.

I’ve temporarily disabled the collision on the bridge border tiles so I can cross them, but that's obviously not a permanent solution.

I need to figure out how to properly layer the collision shapes so that the bridges function correctly.

I suspect this might involve adjusting the Z-indices or possibly restructuring the collision shapes themselves.

Another thing I noticed is that the decorative elements, like trees, bushes, and houses, are lacking collision.

As a result, the player can walk right through them, which breaks the immersion.

I've got a "Decoration" node in my scene where all these elements reside, and I need to add collision shapes to each of them.

This is going to be a bit tedious, but it's essential for making the overworld feel believable.

I also need to address the out-of-bounds issue.

The player is currently confined to the main hub area, which is good, but I need to make sure they can't accidentally wander off into the void.

I'll need to add collision shapes along the perimeter of the hub to create a solid boundary.

So, to summarize, my immediate to-do list looks like this:

Fix the bridge collision issue: Investigate and resolve the overlap between the bridge border tiles and the water collision shapes.

Add collision shapes to the decoration nodes: Ensure that trees, bushes, houses, and other decorative elements have proper collision.

Create out-of-bounds boundaries: Add collision shapes around the hub perimeter to prevent the player from leaving the playable area.

Despite the challenges, I'm feeling good about the progress.

The overworld is starting to take shape, and the collision detection is a major step forward.

I'm learning a lot about how collision shapes work in Godot, and I'm confident that I'll be able to resolve the remaining issues.

I'm excited to see how the overworld evolves as I continue to add more details and functionality.

Finally! The bridge issue is resolved.

I spent a good chunk of time wrestling with those collision shapes, and I'm relieved to have found a solution.

It was simpler than I initially thought, though. Instead of trying to fine-tune the existing collision shapes, I opted for a more straightforward approach.

I created alternative tiles specifically for the bridge sections. These tiles are visually identical to the original bridge border tiles,

but they don't have any collision shapes attached. I then went through the scene and replaced the original bridge border tiles with these new, collision-free versions.

Now, the player can smoothly cross the bridges without getting stuck or blocked by invisible collision.

It's a bit of a workaround, but it works perfectly for now.

With the bridge issue out of the way, I'm turning my attention to the decorative elements and the cliffs.

The overworld is starting to feel more populated, but it still lacks a sense of depth and solidity.

Adding collision to the trees, bushes, houses, and other decorations is the next priority.

I've already started adding CollisionShape2D nodes to these elements, and it's making a noticeable difference.

The player can no longer walk through them, which adds a much-needed layer of realism.

The cliffs are another area that needs attention.

They're currently just flat textures, but I want them to feel like solid, impassable barriers.

I'll need to add collision shapes to the cliffs as well, and I'm also considering adding some visual depth by using parallax scrolling or layering techniques.

I'm making steady progress, and the overworld is starting to feel more like a cohesive and interactive space.

There's still a lot to do, but I'm feeling good about the direction the project is heading.

Adding the collision and refining the boundaries has been a significant step forward, and I'm excited to see how the overworld evolves as I continue to add more details and functionality.

Okay, so I've reached a point where I need to make some strategic decisions.

While the basic overworld layout and collision detection are mostly complete, there are still some lingering details, like adding collision to all the decorations.

I've got houses and market stalls covered, but things like trees and smaller decorative elements are still lacking collision.

However, time is becoming a significant factor.

The end of term is looming, and I need to prioritize completing the core gameplay mechanics.

Therefore, I've decided to shift my focus to implementing the item pickup and inventory system.

The decorations, though important for immersion, can be addressed later.

Right now, I need to get the fundamental gameplay loop working.

I'm diving straight into programming the item pickup logic and the inventory system.

I've already started sketching out the data structures and UI elements I'll need.

My goal is to create a simple yet functional inventory system with slots for storing items.

I'm also planning to implement a basic pickup mechanism that allows the player to interact with items in the environment and add them to their inventory.

My primary objective is to get the main overworld level fully functional, even if it means sacrificing some of the decorative polish for now.

If I can achieve that, I'll then move on to creating a single dungeon level, applying the same gameplay mechanics and learning experiences from the overworld.

I believe that having at least one complete level will demonstrate my understanding of the concepts and be a sufficient deliverable for the project, especially given the time constraints.

I need to communicate this plan to my lecturer, explaining that while I aimed for a multi-level experience, I've prioritized completing a single, fully functional level to meet the core requirements of the project.

I hope they'll understand the time constraints and appreciate the effort I've put into creating a solid foundation for the game.

I'm feeling a bit pressured by the deadline, but I'm determined to make the most of the remaining time.

I'm confident that I can deliver a compelling gameplay experience, even if it's within a single level.

Let's get this inventory system working!

21/03/2025

Today, I took a deep breath and plunged into the next major phase of this project:

implementing the item pickup and inventory system.

To keep things organized, I created a new branch called items-inventory-implementation.

It feels good to start fresh, knowing this is a critical step towards a functional game.

First, I created a scene called pickup-item.

Inside, I added a Node2D as the root, along with a Sprite2D and a CollisionShape2D as its children.

This scene will represent any item the player can pick up in the world.

I also created a new script called inventory_item.gd.

This script is where the magic happens, where I'll define the properties and behaviors of each item.

It's a bit technical, diving into the nitty-gritty of how Godot handles resources and scripts, but it's essential for a robust system.

To start, I created a new folder in my Product directory called resources.

Inside, I made a resource called gold_coin.tres.

This resource will hold the data for gold coins, including their sprite, name, and any other relevant information.

I connected this resource to the inventory_item.gd script, essentially linking the visual representation to its in-game logic.

Now, I'm starting to work on the inventory slots.

It's a bit of a slog, honestly. I'm definitely feeling the pressure of the crunch.

Time is slipping away, and there's still so much to do. It's frustrating when things don't come together as quickly as I'd like, but I'm trying to stay focused.

More than the technical challenges, I'm battling the mental ones. I'm trying to keep the demotivation at bay.

I don’t want to be shackled by the fear of disappointing others, or the pressure of feeling like this is my last chance to make things right.

I’m trying to focus on the process, on learning and building something I can be proud of, regardless of the outcome.

I need to keep reminding myself that progress, even slow progress, is still progress. I hope that even when things seem difficult, I can make something that will be impactful.

24/03/2025

Alright, diving deeper into the technical details of the pickup item system.

I want to make sure I'm documenting this thoroughly, both for my own understanding and in case I need to backtrack or troubleshoot later.

If I'm making any glaring technical blunders, I hope this log will help me catch them.

In the pickup-item.tscn scene, I started by adding two child nodes: a Sprite2D and a CollisionShape2D.

The Sprite2D is where I've created the visual representation of the gold coin pickup item.

I pulled the gold coin sprite from my assets folder and assigned it as the texture for the Sprite2D.

For the CollisionShape2D, I opted for a simple rectangle shape, as it seemed fitting for a small coin.

I then created a new resource called gold_coin.tres in the Product/Resources folder.

This resource essentially holds the data associated with the gold coin, such as the sprite texture.

I connected this resource to the inventory_item.gd script, which I mentioned earlier.

This script is responsible for managing the properties and behaviors of the coin once it's in the player's inventory.

Now, here's where things got a bit tricky. I noticed that the player wasn't consistently picking up the coin.

I suspect the issue lies with the CollisionShape2D and how it's interacting with the player's collision.

To address this, I'm planning to add another layer of collision detection.

My idea is to create a child Area2D node within the pickup-item scene, and then add another CollisionShape2D as a child of that Area2D.

The Area2D will allow me to detect when the player's collision area overlaps with the coin's pickup area.

I'll then connect a signal from the Area2D to the pickup-item node, triggering the pickup logic when the player enters the area.

Essentially, I'm creating a "pickup zone" around the coin, separate from the coin's physical collision shape.

This should give me more precise control over the pickup interaction.

I'm aware that this might seem a bit convoluted, but I'm hoping it will resolve the issue.

If anything goes wrong, I'm prepared to use git reset to revert to a previous state.

I'm documenting this process here so that I can easily retrace my steps and understand what went wrong if I need to.

I'm hoping that by creating this "pickup zone" using an Area2D that the player will be able to pick up the coin. I'm going to implement this now.

With the deadline looming, I'm bracing for an intense push.

My aim is to dedicate the next five days to finalizing the core gameplay elements.

This includes setting up essential features like equipment management, the user interface, weapon attacks, item dropping, configuring spell systems, enemy encounters, and seamless scene transitions between the main hub and the shop.

I'm planning regular progress check-ins to ensure I'm on the right track and to address any potential issues.

My hope is that this concentrated effort will result in a project that exceeds expectations, potentially earning above-average marks.

I must admit, I'm also experiencing some nervousness regarding the report and the overall evaluation.

However, I'm determined to maintain a positive and optimistic mindset.

This is my first significant project, and I'm aware that I'll likely face some criticism.

I'm hoping to learn from it, even if it's difficult, and use it to improve in the future.

Success! I've finally managed to get the player to successfully pick up the gold coin.

After a bit more debugging and tweaking, I pinpointed the issue and implemented a solution.

The key was in the player's Area2D node.

I connected the area_entered signal from the Area2D to the player's script.

Then, I wrote a function that would trigger the pickup logic when the player's Area2D overlapped with the coin's Area2D.

To ensure the player could interact with the coin, I also adjusted the collision layers and masks.

In the player's CollisionShape2D, I set the layer to "Player" and the mask to "PickupItem".

This effectively tells the game that the player should only interact with objects on the "PickupItem" layer.

The coin, of course, is on that layer.

This now means that the player can walk up to the gold coin, and the pickup logic is triggered.

This is a significant milestone, as it lays the foundation for all future pickup interactions.

I'm treating the gold coin as a standard template, which I can adapt for other items.

With the pickup functionality now working, I'm eager to move on to the inventory slot and other inventory-related mechanics.

Hopefully, I can make significant progress on that front in the coming days.

I am feeling a bit relieved that the pick up item works, and I am going to try to do the inventory slot and inventory items soon.

Hopefully it goes well.

Furthermore, I focused heavily on building the inventory slot system.

I created a new scene called inventory_slot.tscn, which will serve as the visual representation of a single inventory slot.

The root node is a Panel, which will act as the background for the slot.

I then added four child nodes: NinePatchRect, MenuButton, CenterContainer, and TextureRect.

The NinePatchRect is used to display the background texture of the inventory slot, which I imported as a "fast box" texture.

This will give each slot a distinct visual appearance.

The TextureRect is used to display the item that is currently occupying the slot.

I adjusted the aspect ratio of the TextureRect to ensure that the items are displayed correctly within the slot.

The MenuButton and CenterContainer are placeholders for future functionality, such as item interaction and quantity display.

I've also imported a large number of assets for various UI elements, including dialogue boxes, inventory screens, and other visual components.

While I'm aware that some of these assets may not be needed in the final version of the game, I'm keeping them for now as a resource library.

I'll likely delete the unused assets later to keep the project organized.

I'm feeling more confident with the development pace now that I've made progress on the inventory system.

My primary goal remains to complete the first level of the game.

If time permits, I'll consider adding additional content or features, depending on the deadline and the flexibility I have.

06/04/2025

Today marked a significant step forward in the development of my project.

I spent a good portion of the day fleshing out the fundamental structure of the inventory system.

My primary focus was on visually representing individual inventory slots, and I opted for a hierarchical approach within the scene tree.

I began by creating a base structure for each slot.

This involved adding several child nodes, each serving a specific visual and potentially functional purpose.

I incorporated nodes that will clearly display the price of an item, ensuring this crucial information is readily available to the player.

Alongside this, I added nodes dedicated to displaying the name of the item occupying the slot, providing immediate identification.

To handle situations where multiple identical items might be present, I also implemented a stack label.

This will dynamically update to reflect the quantity of the item within that particular slot.

Finally, I included a button label within the slot structure.

While its immediate purpose might be visual feedback or a placeholder, it suggests future interactivity, perhaps for selecting, using, or examining the item.

Beyond the visual elements, I also started laying the groundwork for the underlying logic.

I created a script named inventory_slot.gd.

Within this script, I declared several onready variables.

These will serve as direct references to the child nodes I previously added in the scene.

This approach, leveraging Godot's signal/slot system, ensures efficient access to these visual elements from the script.

Furthermore, I utilized the @export keyword (or potentially export depending on the Godot version) to expose certain properties of the inventory_slot script within the editor's Inspector panel.

This will likely include a Vorte texture, which I envision as the visual representation of an empty slot or a background element.

I also anticipate using nine-patch textures (likely configured through a NinePatchRect node, though not explicitly mentioned) for the background of the slot, allowing for flexible resizing without distortion.

Following the creation of this foundational inventory_slot scene and script, I proceeded to instantiate and add several instances of these slots as child nodes to a higher-level inventory container (though the exact parent node wasn't specified).

Crucially, I then imported (likely using get_node() within the parent's script) references to these newly added inventory_slot nodes into the parent script.

This establishes the necessary connection, allowing the parent script to manage and interact with the individual inventory slots.

This feels like a solid initial step in building out the inventory system.

By focusing on creating reusable and well-structured slot elements, I've laid a foundation that should be relatively easy to expand upon.

The use of onready variables and potentially exported properties in the inventory_slot script will streamline the process of populating and updating the visual information within each slot.

This initial setup facilitates the next stage, where I can begin implementing the core logic for adding, removing, and managing items within these slots.

I'm optimistic about moving forward and implementing the actual item management in the near future.

While working on the visual representation of the inventory slots within the 2D editor, I noticed an issue.

Specifically, the initial setup resulted in the creation of twelve individual inventory slots directly within the inventory_slot.tscn scene itself.

This was not the intended approach, as inventory_slot.tscn should serve as a template for individual slots, not the container for multiple instances.

This oversight needs correction to ensure proper modularity and efficient management of inventory space.

Furthermore, I encountered an error related to the absence of a "Popper menu".

This suggests a planned UI element for potential contextual actions or information display related to inventory items.

However, this component hasn't been implemented yet, and I've decided to address it in a later stage to maintain focus on the core slot functionality.

Despite these minor hurdles, I moved forward by creating a new scene called inventory_ui.tscn.

This scene will serve as the primary container for the entire inventory UI.

As the root node of this scene, I added a CanvasLayer node.

CanvasLayer is a crucial node in Godot for UI elements, as it allows them to be rendered on a separate canvas, independent of the game world's camera and scaling.

This ensures that the inventory UI remains consistently sized and positioned on the screen, regardless of the viewport or camera settings.

This CanvasLayer will act as the drawing surface upon which all inventory-related UI elements, including the individual inventory slots, will be placed and managed.

This establishes a dedicated space for the UI, separating it cleanly from the game world and paving the way for further UI development.

More updates to come as I refine the slot instantiation and begin populating the inventory UI.

07/04/2025

Continued work on the inventory_ui.tscn scene today, focusing on its initial structure and basic visibility toggling.

The root CanvasLayer in the inventory_ui now has a ColorRect as a direct child.

This ColorRect, likely set to a red color, serves as a simple background or visual indicator for the inventory screen at this early stage of development.

Nested under the ColorRect is a MarginContainer.

MarginContainer nodes are essential for UI layout in Godot, providing consistent spacing and padding around their child elements.

This ensures that the content within the inventory screen isn't directly touching the edges, improving visual appeal and readability.

Within the MarginContainer, I've established a further nested structure.

This includes a NinePatchRect, which will likely be used to create a resizable background for the main inventory area without pixelation.

The use of a nine-patch texture allows for flexible scaling to accommodate different amounts of content or screen sizes.

Also within the MarginContainer is a VBoxContainer.

VBoxContainer nodes arrange their children vertically, making them ideal for stacking elements like the individual inventory slots.

To implement the basic functionality of showing and hiding the inventory screen, I created a script named inventory.gd

(referred to as "red scripts" in the original note, likely a temporary naming convention).

This script is attached to the root CanvasLayer of the inventory_ui scene.

Within this script, I've connected an input action – specifically, pressing the "Tab" key – to toggle the visibility of the inventory_ui.

This is a common and intuitive control scheme for accessing inventory in many top-down 2D games.

Interestingly, the previous implementation with 30 pre-existing inventory slots, which were toggled with a button pop-up, is still present in the project.

This suggests a potential transition or refactoring process where the new inventory_ui is gradually replacing the older system.

The inventory.gd script likely contains a function, perhaps named toggle_visibility, which is called when the Tab key is pressed.

This function would then manipulate the visible property of the CanvasLayer (or a relevant child node) to show or hide the entire inventory UI.

Looking ahead, the next steps involve populating this basic UI structure with the actual inventory items.

This will likely involve instantiating and adding the individual inventory_slot scenes as children of the VBoxContainer.

I also need to implement the logic for adding items to the inventory, handling item stacking to efficiently represent multiple identical items,

and visually presenting these items within the UI slots.

Beyond the core inventory, the project roadmap includes the development of weapon and resource management systems.

Finally, the goal is to integrate these systems into the action scene, allowing the player to equip weapons and utilize resources within the game world.

The current focus on establishing a clean and functional inventory UI is a crucial stepping stone towards these more complex gameplay mechanics in this top-down 2D game.

To provide a clearer picture, I've included snippets of the core scripts I've been working on.

First, the InventoryUI.gd script, attached to the root CanvasLayer of the inventory UI scene:

```
extends CanvasLayer

class_name InventoryUI

func toggle():
	visible = !visible
```

This script is straightforward but crucial.

It defines the InventoryUI class and implements the toggle() function.

This function simply inverts the visible property of the CanvasLayer, effectively showing or hiding the entire inventory UI on the screen.

Next, the InventorySlot.gd script, which governs the behavior of each individual inventory slot (likely instantiated within the VBoxContainer of the inventory_ui):

```
extends VBoxContainer

class_name InventorySlot

var is_empty = true
```

```
	var is_selected = false


	@export var single_button_press = false

	@export var starting_texture: Texture

	@export var start_label: String

	@onready var texture_rect: TextureRect =
$NinePatchRect/MenuButton/CenterContainer/TextureRect

	@onready var name_label: Label = $NameLabel

	@onready var stack_label: Label = $NinePatchRect/StacksLabel

	@onready var on_click_button: Button = $NinePatchRect/OnClickbutton

	@onready var price_label: Label = $PriceLabel

	@onready var menu_button: MenuButton = $NinePatchRect/MenuButton


	var slot_to_equip = "NotEquipable"


	func _ready() -> void:

		if starting_texture != null:

		texture_rect.texture = starting_texture


		if start_label != null:

			name_label.text = start_label


		menu_button.disable = single_button_press

		on_click_button.disabled = !single_button_press

		on_click_button.visible = single_button_press
```

```
        var popup_menu = menu_button.get_popup()
        popup_menu.id_pressed.connect(on_popup_menu_item_pressed)


    func on_popup_menu_item_pressed(id: int):
        print_debug(id)
```

This script manages the visual elements within each slot.

It declares variables to track if the slot is empty or selected, and exports properties like single_button_press to control the UI interaction style (either a direct button or a menu).

@onready variables efficiently retrieve references to the various child nodes within the InventorySlot scene (TextureRect for the item icon, Labels for name, stack, and price, and Buttons for interaction).

The _ready() function initializes the slot's visual appearance based on the exported starting_texture and start_label.

It also configures the visibility and enabled state of the on_click_button and sets up the PopupMenu associated with the MenuButton, connecting the id_pressed signal to the on_popup_menu_item_pressed function for handling menu item selections.

The InventoryItem.gd script defines the data structure for individual inventory items:

```
    extends Resource
    class_name InventoryItem
    var stacks = 1
```

```
	@export_enum("Right_Hand", "Left_Hand", "Potions", "NotEquipable") var slot_type: String = "NotEquipable"

	@export var ground_collision_shape: RectangleShape2D

	@export var name: String = ""

	@export var texture: Texture2D

	@export var side_texture: Texture2D

	@export var max_stack: int

	@export var price: int
```

y extending Resource, InventoryItem instances can be easily created and stored, either within the game world or within the inventory data structure itself.

It includes properties such as stacks to manage item quantity, an @export_enum for slot_type to define where the item can be equipped (a common design pattern for equipment slots in RPGs), a ground_collision_shape for when the item is dropped in the world, a descriptive name, primary and potentially secondary textures (texture and side_texture), max_stack to limit the number of identical items in a single slot, and the item's price.

Finally, the Inventory.gd script (likely attached to a persistent game manager or the player node) handles the overall inventory logic and UI interaction:

```
	extends Node

	class_name Inventory

	@onready var inventory_ui: InventoryUI = $"../InventoryUI"

	func _input(event: InputEvent) -> void:

		if Input.is_action_just_pressed("toggle_inventory"):

			inventory_ui.toggle()
```

This script retrieves a reference to the InventoryUI node using @onready.

The _input() function listens for specific input events.

In this case, it checks if the "toggle_inventory" action (configured in the Godot project settings) has just been pressed.

If so, it calls the toggle() function on the inventory_ui instance, making the inventory screen appear or disappear. This is a standard approach for handling player input and triggering UI changes in response to specific actions in a top-down 2D game.

And the PickUpItem.gd script, attached to interactable items in the game world:

```
extends Area2D
class_name PickUpItem


@export var inventory_item: InventoryItem
@onready var sprite_2d: Sprite2D = $Sprite2D
@onready var collision_shape_2d: CollisionShape2D = $CollisionShape2D
# Called when the node enters the scene tree for the first time.
func _ready() -> void:
	sprite_2d.texture = inventory_item.texture
	collision_shape_2d.shape = inventory_item.ground_collision_shape
```

This script defines how items in the game world can be picked up.

It exports an InventoryItem resource, allowing each pick-up item instance in the scene to be associated with specific item data.

In the _ready() function, it sets the Sprite2D's texture and the CollisionShape2D's shape based on the properties of the assigned inventory_item,

visually representing the item in the game world and defining its interaction area.

These scripts collectively form the basic framework for the inventory system, handling UI presentation, individual slot behavior,

item data, and player interaction for toggling the inventory screen and defining pick-up items.

The next steps will involve connecting these pieces to enable adding items to the inventory, managing stacks, and displaying the items within the UI slots.

I implemented the code responsible for dynamically generating the individual inventory slots within the inventory_ui.tscn scene.

This involved adding the following code block to the InventoryUI.gd script:

```
	@onready var grid_container: GridContainer = %GridContainer

	const INVENTORY_SLOT_SCENE = 
preload("res://Product/Scenes/UI/inventory_slot.tscn")

	@export var size = 8

	@export var colums = 4


	func _ready():

		grid_container.columns = colums
```

```
    for i in size:
        var inventory_slot = INVENTORY_SLOT_SCENE.instantiate()
        grid_container.add_child(inventory_slot)
```

Here's a breakdown of what this code achieves:

First, it uses @onready to get a reference to the GridContainer node within the inventory_ui scene, which I've named "GridContainer" (indicated by the % symbol).

GridContainer is an excellent choice for arranging inventory slots in a structured grid layout, common in many top-down 2D games.

Next, it uses preload() to efficiently load the inventory_slot.tscn scene into a constant named INVENTORY_SLOT_SCENE.

Preloading optimizes performance by loading the scene into memory when the InventoryUI script is loaded, rather than instantiating it multiple times on the fly.

I've also added two @export variables: size and colums.

size determines the total number of inventory slots to be created, and colums dictates how many slots will be arranged in each row of the grid. Exposing these variables in the Inspector allows for easy adjustment of the inventory grid dimensions without modifying the code directly, a good practice for game design iteration.

The _ready() function is called once when the InventoryUI node enters the scene tree.

Inside this function, I first set the columns property of the grid_container to the value of the exported colums variable, establishing the desired grid width.

Then, a for loop iterates size number of times.

In each iteration, it instantiates a new instance of the inventory_slot.tscn scene using INVENTORY_SLOT_SCENE.instantiate().

This creates a unique copy of the inventory slot template.

Immediately after instantiation, grid_container.add_child(inventory_slot) adds this newly created slot as a child of the GridContainer. This automatically arranges the slots in the defined grid layout.

The immediate result of adding this code is that when I run the game and press the Tab button to toggle the inventory UI,

it now appears populated with the specified number of individual inventory slots arranged in a grid. This is a significant step towards a functional inventory display.

Regarding previous issues, I recalled an error that occurred during initial testing related to the menu_button within the inventory_slot scene.

To resolve this for the time being and allow the game to run without crashing, I commented out the line menu_button.disable = single_button_press in the InventorySlot.gd script.

While this might temporarily disable the intended behavior of the menu button, it allows me to continue focusing on other core inventory functionalities.

I will revisit and properly address this commented-out line later.

With the inventory UI now visually populating with slots, my next priority is to implement the logic for adding items to these slots and handling stackable items.

This will involve connecting the item pick-up system with the inventory data and updating the UI accordingly.

10/04/2025

Today's focus was on a crucial aspect of inventory management: handling stackable items, like our ubiquitous gold coins.

The goal was to allow players to collect multiple instances of the same item and have them occupy a single inventory slot, displaying a quantity.

To achieve this, I revisited the Inventory.gd script and expanded its functionality.

First, I looked at the _on_area_2d_area_entered function (presumably within a player or inventory controller script), specifically the condition for picking up items:

```
if area is PickUpItem:


inventory.add_item(area.inventory_item, area.stacks)

area.queue_free()
```

This snippet demonstrates the initial interaction with a PickUpItem in the game world.

When the player's pickup area overlaps with a PickUpItem, it calls the add_item function in the Inventory script, passing the inventory_item resource associated with the pickup and the number of stacks it represents.

Crucially, after the item is added to the inventory, the PickUpItem node is freed from the scene using queue_free(), removing it from the game world.

Now, let's delve into the Inventory.gd script and the logic for handling stackable items:

```
extends Node

class_name Inventory

@onready var inventory_ui: InventoryUI = $"../InventoryUI"

@export var items: Array[InventoryItem] = []


func _input(event: InputEvent) -> void:

 if Input.is_action_just_pressed("toggle_inventory"):

        inventory_ui.toggle()


func add_item(item: InventoryItem, stacks: int):

  if stacks && item.max_stack > 1:

        add_stackable_item_to_inventory(item, stacks)

  else:

        items.append(item)

        # TODO: update ui


func add_stackable_item_to_inventory(item: InventoryItem, stacks: int):

  var item_index = -1

  for i in items.size():

            if items[i] != null and items[i].name == item.name:

                    item_index = i


 if item_index != -1:
        var inventory_item = items[item_index]


        if inventory_item.stacks + stacks <= item.max_stack:
            inventory_item.stacks += stacks
            # TODO: update player_ui
        else:
            var stacks_diff = inventory_item.stacks + stacks -
item.max_stack
            var additional_inventory_item =
inventory_item.duplicate(true)
            inventory_item.stacks = item.max_stack
            # TODO: update player_ui
            additional_inventory_item.stacks = stacks_diff
            items.append(additional_inventory_item)
            # TODO: update player_ui


    else:
        item.stacks = stacks
        items.append(item)
        # TODO: update player_ui
```

The add_item function is the entry point for adding items to the inventory.

It first checks if the stacks count is greater than zero and if the item's max_stack property (defined in the InventoryItem resource) is greater than 1.

This condition identifies stackable items.

If an item is stackable, it calls the add_stackable_item_to_inventory function; otherwise, it simply appends the item to the items array (for non-stackable items).

A crucial # TODO: update ui comment highlights the need to refresh the inventory UI whenever the items array is modified.

The add_stackable_item_to_inventory function handles the core stacking logic.

It iterates through the existing items in the inventory to find if an item with the same name already exists.

If a matching item is found (indicated by a non-negative item_index):

It retrieves the existing inventory_item.

It then checks if adding the new stacks to the existing item's stacks would exceed the item's max_stack.

If not, it simply adds the stacks to the existing item's count and again, a # TODO: update player_ui reminds us to update the visual representation in the inventory slot.

If adding the stacks would exceed the max_stack, it calculates the stacks_diff – the number of items that would overflow.

It then duplicates the existing inventory_item using duplicate(true) to create a new additional_inventory_item (the true argument performs a deep copy, ensuring all properties are copied).

The original inventory_item's stacks are set to max_stack, and the additional_inventory_item's stacks are set to the stacks_diff.

Finally, the additional_inventory_item is appended to the items array, effectively creating a new stack in a separate inventory slot if the current stack is full.

Another # TODO: update player_ui is essential here.

If no existing item with the same name is found in the inventory, it means this is the first instance of this stackable item being added.

In this case, the item's stacks property is set to the incoming stacks value, and the item is appended to the items array.

Again, the UI needs to be updated.

To test this stacking mechanism, I duplicated the PickUpItem nodes in my game world.

I then modified their stacks property within the Inspector to represent different quantities: one with a stacks value of 12, another with 88, and a third with 20.

This allowed me to simulate picking up varying amounts of the same stackable item (like gold coins) and observe how the inventory system handles them.

During development, I encountered and resolved a couple of syntax errors.

I mistakenly typed inventory.add.item(...) instead of the correct inventory.add_item(...).

Additionally, I incorrectly used item.append(...) when I should have used items.append(...) to add elements to the items array (which is the inventory itself).

These small errors highlight the importance of careful syntax and understanding the context of variables and functions.

With this implementation, the inventory system can now correctly handle stackable items,

consolidating them into single slots and creating new slots when the maximum stack size is reached.

The next crucial step is to ensure the InventoryUI accurately reflects these changes whenever items are added or their stack counts are modified.

This will involve updating the labels within the InventorySlot scenes to display the current stack size.

Today's work on stackable items felt like another solid step forward, reinforcing my belief that even as a solo developer with limited resources, a functional game, even if just a single level, is within reach.

It strikes me that many development teams, whether crafting AAA titles or working in large independent studios, often begin by ensuring they have one polished and engaging level ready.

Considering that I am navigating this endeavor alone, utilizing public domain assets, and this project serves as my introduction to both game development and the Godot engine, the progress feels particularly rewarding.

My workspace, an old laptop with constrained storage, presents its own set of challenges, though I offer this not as a complaint but as a description of my current reality.

The sedentary nature of coding, the prolonged periods spent staring at a screen, certainly aren't conducive to an ideal workflow.

To counter this, I've found solace and a change of scenery by working in a public cafe.

There, I can punctuate coding sessions with slow sips of coffee, allowing my gaze to wander and observe the world around me, providing moments to contemplate life beyond graduation, my aspirations, and dreams, all while the murmur of strangers' conversations fills the background.

For those more intense coding sessions, particularly while implementing the stackable item logic within the inventory script, I immerse myself in "The Autobiography of Malcolm X" as narrated by Laurence Fishburne, allowing his powerful story to occupy my mind while my fingers fly across the keyboard.

I sincerely hope this project serves as a genuine introduction to the broader world of software development, a field I am eager to explore further.

A significant personal goal is to cultivate greater confidence, a quality that sometimes eludes me, though I tend to mask this with a composed exterior and quiet demeanor.

Furthermore, my acknowledged tendency towards a shorter attention span, something my parents have often pointed out, is another personal hurdle I am consciously working to overcome.

Please understand that these reflections are not rooted in any feelings of depression or mental health struggles; that is not the impression I intend to convey.

Instead, these are observations made when I look inward, identifying areas for personal growth and development in the future.

For now, however, my immediate focus returns to the game itself, specifically on the next crucial task: presenting the items currently held within the inventory in a clear and user-friendly way within the Inventory UI.

Mohammad Ibrahim Salman
101029175

16/04/2025

Today's development focused on bridging the gap between the inventory data and its visual representation within the user interface.

The primary goal was to allow the player to see the items they collect, observe their stacking behavior with a displayed quantity,

and provide basic interaction options like dropping or equipping.

To achieve this, I implemented changes in both the InventoryUI.gd and InventorySlot.gd scripts.

Within the InventoryUI.gd script, I added the following add_item function:

```
func add_item(item: InventoryItem):
    var slots = grid_container.get_children().filter(func(slot): return slot.is_empty)
    var first_empty_slot = slots.front() as InventorySlot
    first_empty_slot.add_item(item)
```

This function is responsible for taking an InventoryItem and finding the first available empty slot in the GridContainer.

It retrieves all child nodes of the grid_container and filters them to select only those where the is_empty property (defined in InventorySlot.gd) is true.

It then takes the first empty slot from this filtered list and calls its own add_item function, passing the InventoryItem to it for visual presentation.

Correspondingly, I modified the add_item function within the InventorySlot.gd script:

```
func add_item(item: InventoryItem):
    if item.slot_type != "NotEquipable":
        var popup_menu: PopupMenu = menu_button.get_popup()
        var equip_slot_array_name =
item.slot_type.to_lower().split("_")
        var equip_slot_name = " ".join(equip_slot_array_name)
        slot_to_equip = item.slot_type
        popup_menu.set_item_text(0, "Equip to " + equip_slot_name)
    is_empty = false
    menu_button.disabled = false
    texture_rect.texture = item.texture
    name_label.text = item.name
    if item.stacks < 2:
        return
    stacks_label.text = str(item.stacks)
```

This function in InventorySlot.gd is now responsible for updating the visual elements of a specific inventory slot based on the InventoryItem it receives.

First, it checks if the item's slot_type is not "NotEquipable".

If it is an equippable item, it retrieves the PopupMenu associated with the menu_button.

It then processes the slot_type string (e.g., "Right_Hand") to create a more readable equip slot name ("right hand").

This name is then used to set the text of the first item (index 0) in the PopupMenu to "Equip to [slot name]", providing the "equip" option.

The slot_to_equip variable is also updated with the item's slot_type.

Next, it sets the is_empty flag of the slot to false, indicating that it now contains an item, and enables the menu_button, allowing for interaction (like dropping or equipping).

The texture_rect's texture is set to the item's texture, displaying the item's icon.

The name_label's text is updated with the item's name.

A check is performed to see if the item's stacks count is less than 2. If it is, the function returns, preventing the stack label from being displayed for single items.

Finally, if the stacks count is 2 or greater, the stacks_label's text is set to the string representation of the item's stacks value,

visually showing the quantity of stacked items like gold coins.

During this process, I encountered and resolved a few syntax errors, a common part of the development cycle.

Specifically, I initially had an issue with the set_item_text function in PopupMenu,

where I incorrectly provided three arguments instead of the expected two (the index and the text).

Removing the extra comma resolved this.

Aside from the scripting, I made a minor organizational adjustment by creating a separate folder for UI resources.

This allows for quicker loading of assets like the game's font, improving efficiency.

It feels like my understanding of the Godot engine is deepening with each step. The way scenes, scripts, and signals interact is becoming more intuitive.

With the basic presentation of inventory items now functional,

where collected coins are displayed with their stack count and equippable items offer an "Equip" option,

I feel like I'm heading in the right direction.

My next immediate focus will be on implementing the weapon management system, building upon this foundational inventory framework.

Building upon the visual presentation of items, I've now implemented a crucial update to ensure the inventory UI dynamically reflects the items the player collects and their stacking quantities.

With the newly implemented update_stack_at_slot_index function (though the code for this function wasn't explicitly provided in the prompt, its functionality is clear),

the inventory UI now accurately updates the stack count displayed in each slot as the player picks up more of the same stackable item.

This means that the player can indeed collect and stack up to 100 gold coins (or any item with a max_stack of 100) within a single inventory slot,

with the UI label reflecting the current quantity.

This dynamic updating of the UI provides immediate and clear feedback to the player regarding their collected resources, enhancing the gameplay experience.

With this core inventory functionality now largely in place, my attention is shifting towards the next significant phase of development: combat.

I've organized the upcoming tasks related to combat under the "items-inventory-implementation" branch in my version control system.

This organization includes several key areas: the creation and integration of weapon items, the management of resources relevant to combat (such as ammunition or mana),

the mechanics for equipping weapons, and the design and implementation of the on-screen UI elements necessary for displaying combat information.

As I move into the more intricate aspects of attack animations, I plan to create a separate branch specifically for that purpose.

This will ensure a clean and manageable development process, allowing me to focus solely on animation without disrupting the already established inventory and item systems.

Once the attack animations are in a satisfactory state and thoroughly tested, I will merge the "items-inventory-implementation" branch with the main branch of the project and then update the repository,

integrating all the new combat-related features.

This structured approach to development, utilizing branching and merging, will help maintain code stability and facilitate collaboration (even though I am currently working solo).

The transition to combat mechanics marks an exciting new chapter in this project, and I am eager to begin implementing weapon items and the initial stages of the combat system.

17/04/2025

Today's progress marks the exciting first step into implementing combat mechanics, specifically by introducing a weapon item: the sword, and ensuring it can be represented within the inventory.

To define the properties and behavior of weapon items, I created a new script named WeaponItem.gd, which extends the Resource class:

```
 extends Resource

   class_name WeaponItem

  @export var in_hand_texture: Texture
  @export var side_in_hand_texture: Texture
  @export var collision_shape: RectangleShape2D
  @export_enum("Melee", "Ranged", "Magic") var attack_type: String
```

```
@export_group("Attachment_position")
@export var left_attachment_position: Vector2
@export var right_attachment_position: Vector2
@export var front_attachment_position: Vector2
@export var back_attachment_position: Vector2
@export_group("")

@export_group("Rotation")
@export var left_rotation: int
@export var right_rotation: int
@export var front_rotation: int
@export var back_rotation: int
@export_group("")

@export_group("Z index")
@export var left_z_index: int
@export var right_z_index: int
@export var front_z_index: int
@export var back_z_index: int
@export_group("")

func get_data_for_direction(direction: String):
	match direction:
		"left":
```

```
            return{
                    "attachment_position": left_attachment_position,
                    "rotation": left_rotation,
                    "z_index": left_z_index
            }
    "right":
            return{
                    "attachment_position": right_attachment_position,
                    "rotation": right_rotation,
                    "z_index": right_z_index
            }
    "front":
            return{
                    "attachment_position": front_attachment_position,
                    "roatation": front_rotation,
                    "z_index": front_z_index
            }
    "back":
            return{
                    "attachment_position": back_attachment_position,
                    "rotation": back_rotation,
                    "z_index": back_z_index
            }
```

This script defines the WeaponItem class, inheriting from Resource, which makes it suitable for creating reusable weapon data assets within the Godot editor.

It exports several variables that define the visual and functional aspects of a weapon:

in_hand_texture: This Texture will be used to display the sword (or any weapon) when it is held by the player.

side_in_hand_texture: This Texture might be used for a different visual representation of the weapon when viewed from the side, potentially for 2D sprite direction handling.

collision_shape: This RectangleShape2D defines the physical boundaries of the weapon, crucial for hit detection during combat. By including it directly in the resource, each weapon can have a unique hitbox.

attack_type: This uses @export_enum to restrict the possible values to "Melee", "Ranged", or "Magic", categorizing the weapon's attack style. This is important for implementing different combat behaviors later.

The script then defines three @export_group sections: "Attachment_position", "Rotation", and "Z index". These groups organize related properties within the Inspector panel, making it easier to manage the weapon's visual placement and layering relative to the player sprite for different facing directions.

Within each of these groups, there are Vector2 variables for left, right, front, and back attachment positions. These determine where the weapon sprite should be positioned relative to the player's hand or body when equipped and facing a particular direction.

Similarly, integer variables for left, right, front, and back rotations define the weapon's visual orientation for each direction.

Finally, integer variables for left, right, front, and back Z indices control the rendering order (layering) of the weapon sprite relative to other sprites, ensuring it appears correctly in front or behind the player depending on the viewing angle.

The get_data_for_direction function takes a direction string as input and uses a match statement (similar to a switch statement in other languages) to return a dictionary containing the attachment position, rotation,

and Z index specific to that direction.

This function encapsulates the logic for retrieving the correct visual transformation data based on the player's facing direction,

making it easier to handle directional weapon rendering.

To see the sword in action, I created a WeaponItem resource in the editor, assigned it a texture for its in-hand appearance and defined its collision shape.

By adding an instance of this WeaponItem to the player's inventory (for now, likely done manually for testing), the groundwork is laid for it to be displayed.

I spent some time tweaking the Inspector tabs to ensure these new weapon properties were organized logically and easy to understand, reflecting good software design principles focused on user experience within the development environment itself.

I also addressed a few minor syntax errors that inevitably arise during coding.

My next immediate goal is to dive deeper into scripting the player's ability to equip this sword as their active weapon.

This will likely involve modifying the player's control script to handle equipping actions and visually attaching the weapon sprite to the player's hand based on their current facing direction,

utilizing the attachment positions and rotations defined in the WeaponItem resource.

Following the successful implementation of weapon equipping, I will create a new branch in my version control system,

specifically dedicated to developing the attack animations for the sword. This branched approach will allow me to iterate on the visual aspects of combat without destabilizing the core inventory and item management systems that I've been building.

Once the animations are satisfactory, I will merge this animation branch back into the main development branch and update the repository, integrating the initial combat mechanics into the game.

This step into weapon implementation feels significant. It marks the transition from purely inventory management towards dynamic gameplay interactions.

The structure I've established with the WeaponItem resource, including directional visual data, reflects common practices in 2D game development for handling sprite orientation and attachments.

As a solo developer, organizing these systems clearly is crucial for maintainability and future expansion. The cultural aspect of game development often involves sharing these kinds of reusable data structures within teams to ensure consistency and efficiency.

Even working alone, adopting these established patterns helps in thinking like a professional software developer.

The lifestyle of a game developer often involves this iterative process of designing data structures, implementing logic, and then visually realizing those systems in the game world. Seeing the sword appear in the inventory is a tangible reward for the abstract work of writing code and defining data.

19/04/2025

Today's development focused on enabling the player to equip weapons from their inventory,

a crucial step towards implementing a functional combat system.

The process involved modifications to several scripts,

primarily centered around handling the equipping action and communicating it between the inventory UI and the core inventory logic.

First, I declared a new signal in InventoryUI.gd: signal equip_item(index: int, slot_to_equip: String)

This signal, equip_item, is emitted by the InventoryUI when the player selects the "Equip" option from an item's context menu.

It carries two important pieces of data: the index of the inventory slot containing the item to be equipped, and the slot_to_equip string,

which indicates the intended equipment slot (e.g., "Right_Hand", "Left_Hand").

Next, I edited the on_popup_menu_item_pressed function within InventorySlot.gd.

This function now handles different actions based on the id of the menu item pressed in the PopupMenu.

Instead of the previous generic print_debug(id), I've implemented a match statement to provide specific behavior for different menu options.

I've also added a "Drop" and "Examine" option, even if their functionality is not fully implemented yet.

When the "Equip to ..." option (assumed to have an ID of 0) is selected, the code checks if the item is actually equippable (i.e., slot_to_equip is not "NotEquipable").

If it is, the script retrieves a reference to the InventoryUI node by traversing the scene tree (getting the parent of the parent).

It then emits the equip_item signal, passing the slot's index (obtained using get_index()) and the slot_to_equip string.

The print_debug line is commented out, but kept for debugging purposes.

I've added placeholder comments for the "Drop" and "Examine" options, indicating where their logic should be implemented later.

To handle this signal, I added the following code to Inventory.gd: func _ready() -> void: inventory_ui.equip_item.connect(on_item_equipped) ; and func on_item_equipped(index: int, slot_to_equip: String): ......

In the _ready() function, I connect the inventory_ui's equip_item signal to a new function called on_item_equipped.

This function is called when the equip_item signal is emitted. It receives the slot index and the slot_to_equip string as arguments.

The function retrieves the corresponding item_to_equip from the items array using the provided index.

Currently, it only prints a debug message to confirm the correct item and slot are being processed.

I've added a comment to indicate where the actual logic for equipping the item (attaching it to the player, updating the player's appearance, etc.) will be implemented.

Finally, I deleted a duplicate _input function that was present in the Inventory.gd script.

This series of changes establishes the communication pathway for equipping items.

When the player chooses to equip an item from the inventory UI, the equip_item signal is emitted,

carrying the necessary information to the Inventory script, which then begins the equipping process.

This is a fundamental step in making the combat system functional.

The design pattern of using signals to communicate between different parts of the game (UI and game logic) is a cornerstone of good software development,

promoting modularity and maintainability.

Being game developers, we often work with complex systems, and signals provide a clean way to decouple components.

The choice to use a signal here reflects a design decision to separate the UI's responsibility (handling user interaction) from the game logic's responsibility (managing the game state and applying the effects of actions).

This separation of concerns is a core principle in software engineering and is valuable in any complex project, including game development.

20/04/2025

Today's development session was largely dedicated to resolving a critical error that prevented equipped items from being displayed on the screen.

The error message I encountered during playtesting in the Godot engine was:

"Attempt to call function 'equip_item' in base 'null instance' on a null instance"

This error occurred within the Inventory.gd script, specifically at line 58:

```
 extends Node

 class_name Inventory


@onready var inventory_ui: InventoryUI = $"../InventoryUI"

@onready var on_screen_ui: OnScreenUI = $OnScreenUI
```

```
@export var items: Array[InventoryItem] = []


func _ready() -> void:
	inventory_ui.equip_item.connect(on_item_equipped)


@warning_ignore("unused_parameter")
func _input(event: InputEvent) -> void:
	if Input.is_action_just_pressed("toggle_inventory"):
		inventory_ui.toggle()


func add_item(item: InventoryItem, stacks: int):
	if stacks && item.max_stack > 1:
		add_stackable_item_to_inventory(item, stacks)
	else:
		# Syntext Error
		items.append(item)
		inventory_ui.add_item(item)


func add_stackable_item_to_inventory(item: InventoryItem, stacks: int):
	var item_index = -1
	for i in items.size():
		if items[i] != null and items[i].name == item.name:
			item_index = i
	if item_index != -1:
		var inventory_item = items[item_index]


		if inventory_item.stacks + stacks <= item.max_stack:
			inventory_item.stacks += stacks
			items[item_index] = inventory_item

inventory_ui.update_stack_at_slot_index(inventory_item.stacks, item_index)
		else:
			var stacks_diff = inventory_item.stacks + stacks -
item.max_stack
			var additional_inventory_item =
inventory_item.duplicate(true)
			inventory_item.stacks = item.max_stack

inventory_ui.update_stack_at_slot_index(inventory_item.stacks, item_index)
			additional_inventory_item.stacks = stacks_diff
			# error fixed: syntex error
			items.append(additional_inventory_item)
			inventory_ui.add_item(additional_inventory_item)


	else:
		item.stacks = stacks
		items.append(item)
		inventory_ui.add_item(item)
 func on_item_equipped(index: int, slot_to_equip: String):
     var item_to_equip = items[index]
     on_screen_ui.equip_item(item_to_equip, slot_to_equip) # Line 58
```

This error message, a common one in Godot, indicates that the code was attempting to call the function equip_item on a variable (on_screen_ui) that held a null value.

In essence, the Inventory script couldn't find the OnScreenUI node in the scene, and therefore, on_screen_ui was never assigned, resulting in a null instance.

To provide more context, here are the relevant code snippets from InventoryUI.gd:

```
     extends CanvasLayer
 class_name InventoryUI


 signal equip_item(index: int, slot_to_equip: String)


 @onready var grid_container: GridContainer = %GridContainer
 const INVENTORY_SLOT_SCENE =
preload("res://Product/Scenes/UI/inventory_slot.tscn")


 @export var size = 8
 @export var colums = 4
```

```
func _ready():
	grid_container.columns = colums


	for i in size:
		var inventory_slot = INVENTORY_SLOT_SCENE.instantiate()
		grid_container.add_child(inventory_slot)


		inventory_slot.equip_item.connect(func(slot_to_equip: String):
equip_item.emit(i, slot_to_equip))


func toggle():
	visible = !visible


func add_item(item: InventoryItem):
	var slots = grid_container.get_children().filter(func(slot): return
slot.is_empty)
	var first_empty_slot = slots.front() as InventorySlot
	first_empty_slot.add_item(item)


func update_stack_at_slot_index(stacks_value: int, inventory_slot_index: int):
	if inventory_slot_index == -1:
		return
	var inventory_slot: InventorySlot =
grid_container.get_child(inventory_slot_index)
```

```
        inventory_slot.stacks_label.text = str(stacks_value)
```

And from InventorySlot.gd:

```
        extends VBoxContainer
 class_name InventorySlot


 var is_empty = true
 var is_selected = false
 signal equip_item


 @export var single_button_press = false
 @export var starting_texture: Texture
 @export var start_label: String


 @onready var texture_rect: TextureRect =
$NinePatchRect/MenuButton/CenterContainer/TextureRect
 @onready var name_label: Label = $NameLabel
 @onready var stacks_label: Label = $NinePatchRect/StacksLabel
 @onready var on_click_button: Button = $NinePatchRect/OnClickbutton
 @onready var price_label: Label = $PriceLabel
 @onready var menu_button: MenuButton = $NinePatchRect/MenuButton


 var slot_to_equip = "NotEquipable"


 func _ready() -> void:
```

```
	if starting_texture != null:
		texture_rect.texture = starting_texture

	if start_label != null:
		name_label.text = start_label

	#menu_button.disable = single_button_press
	on_click_button.disabled = !single_button_press

	on_click_button.visible = single_button_press

	var popup_menu = menu_button.get_popup()
	popup_menu.id_pressed.connect(on_popup_menu_item_pressed)

func on_popup_menu_item_pressed(id: int):
	var pressed_menu_item = menu_button.get_popup().get_item_text(id)

	if pressed_menu_item == "Drop":
		#TODO: handle item dropping
		print_debug("DROP")
	elif pressed_menu_item.contains("Equip") && slot_to_equip != 
"NotEquipable":
		equip_item.emit(slot_to_equip)
	#print_debug(id)
 func add_item(item: InventoryItem):
        if item.slot_type != "NotEquipable":
                var popup_menu: PopupMenu = menu_button.get_popup()
                var equip_slot_array_name = item.slot_type.to_lower().split("_")
                var equip_slot_name = " ".join(equip_slot_array_name)
                slot_to_equip = item.slot_type
                # Syntext Error fixed: expected two arguments but got three.
Removed the comma after "Equip to"
                popup_menu.set_item_text(0, "Equip to " + equip_slot_name)


        is_empty = false
        menu_button.disabled = false
        texture_rect.texture = item.texture
        name_label.text = item.name
        if item.stacks < 2:
                return


        stacks_label.text = str(item.stacks)
```

And the relevant OnScreen UI code:

```
        extends VBoxContainer
 class_name OnScreenEquipmentSlot
```

~~~
@onready var slot_label: Label = $SlotLabel

@onready var texture_rect: TextureRect = %TextureRect


@export var slot_name: String


func _ready() -> void:

	slot_label.text = slot_name


func set_equipment_texture(texture: Texture):

	texture_rect.texture = texture
```gd
extends CanvasLayer

class_name OnScreenUI


@onready var right_hand_slot: OnScreenEquipmentSlot =
$MarginContainer/HBoxContainer/RightHandSlot

@onready var left_hand_slot: OnScreenEquipmentSlot =
$MarginContainer/HBoxContainer/LeftHandSlot

@onready var potion_slot: OnScreenEquipmentSlot =
$MarginContainer/HBoxContainer/PotionSlot

@onready var spell_slot: OnScreenEquipmentSlot =
$MarginContainer/HBoxContainer/SpellSlot


@onready var slots_dictionary = {

	"Right_Hand": right_hand_slot,

	"Left_Hand": left_hand_slot,
~~~

```
	"Potions": potion_slot
}


func equip_item(item: InventoryItem, slot_to_equip: String):
	slots_dictionary[slot_to_equip].set_equipment_texture(item.texture)
```

The core issue was a misconfiguration in the scene tree, leading to an incorrect node path.

As the diary entry notes:

"The error occurred because on_screen_ui was assigned using $OnScreenUI, assuming that the OnScreenUI node was a direct child of this Inventory node.

However, at runtime, the node was not found at that path, resulting in a null reference when calling equip_item()."

The solution involved restructuring the scene:

"RESOLUTION: I moved the OnScreenUI node to be an instant child above both InventoryUI and Inventory in the scene tree,

ensuring that it is properly instantiated and accessible at the time Inventory is ready. Now $OnScreenUI correctly points to the node, and the error is resolved."

In essence: The Inventory script uses the @onready keyword to get a reference to the OnScreenUI node.

The $ syntax specifies a relative path within the scene tree. If the OnScreenUI node is not located at the path specified, the @onready variable will be assigned null.

By adjusting the scene tree so that OnScreenUI is a direct child of the node that contains Inventory, the path $OnScreenUI correctly resolves to the OnScreenUI node.

By correcting the scene structure, the Inventory script was able to correctly access the OnScreenUI node and call its equip_item function, allowing the equipped item to be displayed.

The diary entry concludes with the intention to: "and now I'm going to merge the items-inventory-implementation branch with the main, and moving to combat."

This indicates a successful resolution of the issue and a transition to the next development phase: implementing the combat system.

21/04/2025

Today marked a significant milestone in my game development journey.

I successfully merged the items-inventory-implementation branch into the main branch.

This was accomplished without losing any data or negatively impacting the history of previous successful merges.

I paid close attention to potential merge conflicts, particularly in the game.tscn scene file, and carefully resolved these conflicts to ensure the integrity of all item and inventory systems.

After thoroughly confirming the merge's success, I pushed the updated main branch to both GitLab and GitHub.

This action ensured that the entire codebase and its complete commit history are synchronized across both platforms.

I also gained valuable insight into how importing projects from GitLab to GitHub can affect the visibility of branches and commits.

Specifically, I learned how to accurately push all branches and commit history to maintain consistency across different version control systems.

Next, I dove into the exciting task of implementing attack animations for the player character.

I already had a functional AnimatedSprite2D setup in place, with working idle and run states.

I expanded the player.gd script to trigger directional attack animations based on the player's last movement direction.

To prevent the player from moving during attacks, I introduced an is_attacking flag.

This flag effectively locks player movement and prevents switching between idle and run animations while an attack is in progress.

I then connected the animation_finished() signal to a new method.

This method resets the is_attacking flag after each attack animation completes, allowing the player to move again.

I also identified and fixed an issue where the attack animation would loop indefinitely. This was resolved by ensuring that the attack animation's "Loop" property was correctly disabled within the Godot editor.

A thoughtful design question arose during this process: Should the player be able to move while attacking?

After careful consideration of the gameplay feel in classic Zelda-like games, I made a deliberate decision.

I decided that attacks should briefly lock the player in place. This design choice aims to create a more intentional and strategic pace to combat,

emphasizing deliberate actions over frantic button-mashing.

This design philosophy reflects a broader trend in game development, where designers often prioritize player agency and meaningful choices over pure action.

It also touches upon cultural differences in game design, where different regions and player demographics may have varying preferences for game feel and pacing.

What’s next on my development roadmap: I still need to refine the combat system further.

Specifically, I plan to make the attack animation play only when the player has a sword (or another suitable weapon) equipped in their inventory.

Soon, I will also begin to add spells to the player's combat repertoire. This addition of spells will introduce more dynamic and varied combat options, adding depth and complexity to the gameplay.

And of course, the game world will eventually need enemies to fight, which is a whole other exciting challenge!

I started by tackling the challenge of dynamically controlling attack animations based on the weapon's direction.

I've implemented a system to fetch positional and visual data, including rotation, attachment point, and z-index.

This data is retrieved from each weapon's WeaponItem resource using a newly created get_data_for_direction() method.

The data structure within the WeaponItem resource utilizes direction keys such as "left", "right", "front", and "back".

This design choice provides a clean and organized way to map weapon visuals to different attack states.

One issue I encountered was an error related to accessing player.attack_direction. It turned out that this property did not yet exist within the player.gd script.

This led to a discussion about aligning the direction naming convention. Specifically, we considered whether to use "up"/"down" or "front"/"back".

I ultimately decided to convert my internal logic within the player script to use "front" and "back".

This decision was made to ensure consistency and to better match the data structure already established in the WeaponItem resource.

Maintaining this consistency improves code readability and reduces the potential for future errors.

After resolving the direction naming convention, I focused on ensuring that attacks would only register when a weapon, such as the sword, is actually equipped.

I updated the _input() logic within the CombatSystem.gd script to check for the presence of either a right_weapon or left_weapon before allowing any attack animation to play.

This prevents the player from performing empty or ineffective attack animations. I also added a helper function called _set_weapon_pose().

This function is responsible for applying the direction-based visual transformations to the weapon sprite when an attack occurs.

This ensures that the weapon is displayed correctly, with the appropriate rotation and position, during the attack animation.

Finally, I addressed an issue where the sword sprite was always visible, even when the player was not attacking.

I fixed this by initially hiding both the right and left weapon sprites within the _ready() function.

The weapon sprites are now only made visible during the execution of an attack animation. Once the animation completes, the on_attack_animation_finished() function is called.

This function resets the visibility of the weapon sprites and also resets the player's attack state. This approach creates a much cleaner and more immersive visual effect.

The sword now feels like a tangible extension of the player, only appearing when it's actively being used in combat.

The entire flow of the combat system now feels significantly tighter and more polished.

The animations are clean and responsive, there are no more floating swords, and input is correctly restricted based on whether a weapon is equipped.

This improved system enhances the overall feel of the game and provides a more satisfying player experience.

My next goals for the combat system are to add hitboxes to the attack animations.

I also want to implement visual feedback when an attack connects with an enemy.

This could include things like particle effects, screen shake, or temporary changes in enemy appearance.

I've been experimenting with how removing or clearing resources impacts gameplay within the Godot engine.

Specifically, I opened the sword_weapon_item.tres file, located at res://Product/Resources/Weapons/Sword/sword_weapon_item.tres, and cleared the assigned "in-hand" texture and collision shape.

Upon running the game, the sword predictably disappeared, as it was now lacking the necessary visual and physical properties.

These experiments yielded some key observations:

Sprite Texture:

Setting a sprite's texture to null results in the sprite becoming permanently invisible unless a new texture is explicitly assigned.

This highlights the importance of managing resource dependencies and understanding how the engine handles null values in visual components.

Node/Resource Dependency:

Removing a node or resource without carefully checking its dependencies throughout the project can lead to broken functionality in seemingly unrelated areas.

For example, deleting the sword_weapon_item.tres file without considering its connections to equipment logic, combat systems, and other game mechanics can introduce subtle bugs that may be difficult to trace and debug.

This underscores the importance of modular design and dependency management in game development.

Error Handling:

To further investigate the consequences of resource removal, I deleted the .tres resource directly through the Godot inspector and then ran the game to observe the resulting errors.

As expected, this action caused significant issues, demonstrating the critical role these resource files play in the game's structure and behavior.

I immediately reverted this change using Git by executing git restore Product/Resources/Weapons/Sword/sword_weapon_item.tres in Git Bash, effectively recovering the deleted resource.

This experience reinforced the value of version control in safeguarding against accidental data loss and facilitating rapid recovery from experimental changes.

A word of caution:

Deleting .tres files or removing resources from the inspector without thoroughly checking their dependencies may not always cause immediate,

catastrophic crashes. However, such actions can easily break functionality in unexpected and hard-to-find places within the game.

This can lead to a significant increase in debugging time and potentially introduce long-term instability.

Therefore, it's crucial to always double-check where resources are being used before removing them from the inspector or deleting the files altogether.

This principle extends beyond game development and applies to software engineering in general: understanding the interconnectedness of different components is essential for maintaining a stable and robust system.

22/04/2025

Today was a good example of how my day-to-day flows as a solo dev — a mix of building, testing, breaking, and debugging — and somehow it all still feels rewarding.

I spent a big chunk of the day tightening up the item drop system in my Godot project.

I'd already wired up most of the inventory logic — the UI components, the signals, the scripts for managing slots and items — but now I wanted to let the player physically eject items from their inventory into the game world.

It's a small feature on paper, but it ties together a lot of different systems: the inventory backend, the visual UI, the game world's spawning logic, and player input.

I started by defining an item_eject_direction in my player.gd script,

which follows the last movement input, so the item knows which way to pop out.

Then I went into Inventory.gd and added a method called eject_item_into_the_ground(idx) — simple enough: grab the item at the given index, spawn a pickup object, assign it the right data, and throw it into the world with some velocity.

Running the game the first time, everything seemed okay… until I hit the "Drop" button — and boom, an error:

> "Out of bounds get index '0' (on base: 'Array[InventoryItem]')"

The engine highlighted multiple lines in different files — from Inventory.gd to InventoryUI.gd and even InventorySlot.gd. That's where the beauty (and chaos) of interconnected code becomes real.

Every piece was working in isolation, but as soon as they started talking to each other, one invalid index took down the whole interaction.

Debugging it required tracing the signal flow: the slot emitted a drop event, the UI relayed it, the inventory script picked it up, and tried to drop an item that no longer existed.

Classic case of bad timing — I was clearing the item slot before trying to use the item’s data to spawn it in the world.

So I reversed the order of operations: eject first, clear second.

I also added bounds and null checks to be safe — something that should always be second nature but is easy to overlook when you're in a build sprint.

After patching up the logic and re-running the game, the error disappeared.

More importantly, the item now gracefully launches into the world, just like I imagined.

What I love about this process — the game dev loop — is that it feels like problem-solving in motion.

It’s half engineering, half art. Each bug is like a little mystery to unravel, and each fix deepens my understanding of the system I’m building.

It reminds me why I chose this route: to create something that feels alive, even if it’s all just pixels and code underneath.

Culturally, I think game design attracts people who want to build worlds, not just apps. It’s about expression as much as functionality.

And for solo developers like me, this work shapes our whole lifestyle — working in bursts of focus, making space to test and reflect, and constantly pivoting between systems thinking and visual storytelling.

So yeah — today was about item drops.

But really, it was about how even the smallest gameplay action requires harmony between systems. And how debugging is just another form of design — one where your tool is patience, your canvas is the codebase, and your payoff is that sweet moment when it all clicks.

Developing spell configurations.

To break up the monotony a bit, I'm adding a new system for spellcasting, which will be a good learning experience for a beginner in game development.

In the game's user interface, the right-hand side will display short combat actions,

while the left-hand side of the inventory slots will be dedicated to the spells the player can cast.

Initially, I'm focusing on implementing Fire and Ice spells.

I've developed the necessary resources for these spells, including their collision shapes, textures, and item representations.

After importing the assets, such as the textures, in the inspector, I added their individual collision shapes under the resources folders.

This ensures that the spells will appear correctly on screen and interact with the game world.

Now, when the player picks up a spellbook and presses the 'L' key, they can cast the associated spell in any direction.

While enemies aren't yet implemented, I've added animations for both the Fireball and Ice spells.

I've also written some code and scripts for spell configuration.

This includes setting up the properties that will appear in the inspector tab,

as well as the code itself for handling projectile movement, direction, and connecting these behaviors to the spell configuration.

My next steps involve developing the user interface for spellcasting and implementing the spell cooldown mechanism.

This will involve creating visual elements to represent the available spells and their cooldown states, as well as writing the code to manage the timing of spell usage.

This is an exciting step that will bring the magic system to life, and I'm looking forward to the challenges it presents.

When Breakpoints Meet Breakthroughs

Today was one of those classic “why is this breaking again?” kind of days in the dev cave.

Everything started off with what should’ve been a simple tweak to the UI system for spell slots.

I wanted a nice clean way to show or hide the magic UI depending on what kind of weapon the player equips — sword? No magic. Staff? Magic time. Seems straightforward, right?

But as always, reality checked in.

I started by trying to call toggle_spell_slot() and passed it the expected values. But Godot was like, “Breakpoint, my friend,” and stopped the show every time.

I threw in some print statements to track the execution flow. Didn’t help much — breakpoints still hit, and it seemed like the engine was choking on the function call like it hadn’t even registered it existed.

What followed was a bit of rabbit-hole spelunking — double-checking signal connections, refactoring names, staring into the void of InventoryUI.gd, and whispering to the Godot Debugger like it was a haunted Ouija board. At one point, I even suspected a misconfigured signal or a hidden circular reference. Classic overthinking move.

Godot started handing me yellow warning signs like a safety officer on overtime:

"You're shadowing is_visible and is_selected!"

“You’re not using event in _input() — shame on you!”

Nice to know, but not the issue. Just background noise while the real bug threw a tantrum.

I ditched the previous spell slot logic and rewrote the whole thing more functionally:

```
func check_magic_ui_visibility():
 var should_show_magic_ui = (combat_system.left_weapon != null and \
 combat_system.left_weapon.attack_type == "Magic") or \
 (combat_system.right_weapon != null and \
 combat_system.right_weapon.attack_type == "Magic")
 inventory_ui.toggle_spells_ui(should_show_magic_ui)
  if should_show_magic_ui == false:
        on_screen_ui.toggle_spell_slot(false, null)
```

Boom. That cleared the crash.

Everything felt cleaner — no shadowed vars, no mysterious breakpoints, and spell slots now behave with actual logic.

There’s something wild about how debugging code mirrors the rest of life: a mix of persistence, creativity, and sometimes a bit of stubbornness.

One line in the wrong place and the whole mood shifts. Like knocking over a candle in a medieval tavern — suddenly you're doing fire control instead of storytelling.

Game dev is such a beautifully messy art. You're half-engineer, half-author.

A storyteller who needs to know memory management and how to trigger an animation with three boolean flags.

The process feels like jazz sometimes — a bunch of trial and error, tweaks, and "does this work if I just nudge it slightly?"

And I guess that’s what I love. You design spells and swords, yes — but you also design how people feel when they click on something.

That’s culture. That’s meaning. Whether you’re making Wimbledon's Lot or just a little inventory UI, you're shaping digital behavior with human intention.

Debugging isn’t just technical — it’s philosophical.

You stare at broken systems and believe they can be fixed.

You imagine something better and keep poking until the engine listens.

It's like gardening in a digital dimension — a bit messy, but the reward? Magic.

Mood: Tired but satisfied.

Lessons Learned: Don't trust early breakpoints. Shadowing is sneaky. And write UI logic that actually makes sense.

Next Up: Animating the spell slot when a spell is equipped. Maybe with a little whoosh sound, just for flair.

24/04/2025

After what felt like a relentless battle of trial and error, I finally got projectiles working in the game.

Yes — the spells now cast, fly, and even look halfway decent doing so.

It's been a long haul: debugging missing functions, chasing down invalid property accesses, restructuring Player.gd to expose attack_vector correctly,

and making sure the spell system actually talks to the UI, combat system, and player logic.

A few core problems held me back longer than I expected:

My Player script didn't initially expose the directional attack vector properly.

Signals between systems (like cast_active_spell) weren't connecting at first.

A method spell_cooldown_activated was called from SpellSystem, but never defined in OnScreenUI. A classic case of "I swear I added that already."

Many small but critical @onready and signal connection details needed to be just right — and weren't, until they were.

When everything finally clicked together, that first Fireball cast felt like a real moment. That said, not everything’s perfect.

The vertical direction is flipped, meaning spells go up when I mean down and vice versa.

On top of that, the ice projectiles are facing the wrong way horizontally. It's a visual bug, but since it doesn’t block gameplay logic or system functionality,

I’ve chosen to ignore it for now.

I’m on a tight clock, and spells firing at all is a huge milestone. So I’m shelving polish for later.

Next up: Enemy behavior and hitboxes. I’ve branched off to a new feature branch to start prototyping enemy interactions.

Let’s go.

25/04/2025

I've now shifted my focus to developing the enemies for the game.

First, I created two new scenes: one for the enemy itself, and another dedicated to the health system, which will be used for both the player and the enemies.

For the enemy scene, I added several child nodes.

These include animation players for walk, idle, and death animations, ensuring that the enemies have a visual representation of their state.

I also added a collision shape, with its data stored in a resource file, and an Area2D node.

The Area2D will likely be used for detecting when the player or other game elements are within range of the enemy, potentially for attacks or other interactions.

To integrate the health system with the enemy, I created an instanced child scene.

This approach allows me to connect the health system's functionality to each enemy instance.

Additionally, I added a progress bar node to the enemy scene.

This progress bar will visually represent the enemy's remaining health, similar to many other games, providing the player with clear feedback on how much damage the enemy has sustained.

I also wrote scripts for both the health system and the enemy.

In the enemy script, I've preloaded the pickup item scene.

This is intended for use later in the combat system, likely to handle what happens when an enemy is defeated, such as dropping loot or other rewards.

I'm going to break this diary entry here, as I'm eager to continue writing more code and further develop these systems.

I feel like I'm making significant progress, and I'm almost at a point where I'll consider this project complete, for now.

I've continued working on the enemy system, and now the enemies are capable of movement.

Specifically, if markers are placed in a square or rectangular pattern, the enemies will move to each marker in sequence.

They will then loop around the pattern, continuously moving from one marker to the next.

This creates a basic patrol or movement pattern for the enemies.

I've also started implementing collision detection between the player and the enemies.

Now, if the player collides with an enemy, the player will take damage.

This establishes a fundamental element of risk and consequence in the gameplay.

I'm currently working on how the player can defeat the enemies.

My focus is on implementing the core mechanics for combat and projectiles.

This will allow the player to fight back against the enemies and defend themselves.

In the Node inspector tab, there are many functions available.

Specifically, within the player scene, the "right-hand weapon Sprite" and "left-hand weapon Sprite" are children of the 2D scene.

I connected the body_entered signal to the combat system.

This means that when the player's weapon comes into contact with an enemy's hitbox, damage is registered.

However, I have to admit that I encountered some confusion regarding directional terminology.

I was initially unsure about the most appropriate way to label "left," "right," "up," and "down" in relation to the player's weapon sprites.

As a result, there's an inconsistency: the weapon's sprite detection and the actual hand position are sometimes in opposite directions.

While the damage registration works, the visual feedback might be slightly inconsistent.

Unfortunately, at this time, I lack the time to thoroughly polish this aspect of the combat system.

Therefore, I need to move on and quickly develop the last few remaining parts of the project.

The next major task is implementing the merchant section, which I'm planning to create as a separate branch.

I've decided to name this branch "merchant," as I think it sounds more fitting than "shop."

If time permits, I might also develop a startup menu or boot sequence.

However, I'm not optimistic about having enough time for that feature.

So yeah, that's pretty much where the project stands for now.

I've been developing the scene transition system, which is crucial for implementing the merchant functionality.

My plan is to allow the player to enter a building, such as a house, and interact with a merchant to purchase items like health potions.

To achieve this, I'm connecting sub-nodes using Area2D nodes.

I've written two primary scripts for this system.

The first, transition_change, handles the actual scene transition, specifically exiting to the shop scene.

The second script, shop_entrance, is attached to the Area2D and connects to the shop_scene_transition_manager.

This manager, in turn, handles the visual transition effects, such as fading and sliding.

For the Area2D node, I used the Node inspector tab to connect the body_entered signal directly to the area itself.

This facilitates the scene change when the player's body enters the defined area.

In the project settings, I navigated to the "autoload" section under "globals."

There, I added the path to the scene and then added the transition manager node.

This approach provides a scalable way to manage scene transitions in future games, especially when creating multiple levels and indoor scenes.

It's essentially an expansion of Godot's singleton/autoloading feature.

This feature allows you to store global variables, such as player information, and effectively simulate a combined scene tree, acting as a kind of "scene main loop."

I've added some tips and basic information about how this works within Godot.

Finally, I'm moving on to develop the merchant's main scene.

This will involve creating the shop transition, the merchant character, and the basic shopping interface.

This system is a key part of the game.

I developed the merchant system, which allows the player to buy and sell items.

I created two new scripts: shopping_ui.gd and merchant.gd, for the system itself.

I also incorporated the transition change functionality.

In the main game scene, I added a merchant's house (or shop) as a parent node.

The player can enter this shop, which then transitions the player to a new scene representing the shop's interior.

Inside the shop scene, the player can interact with the merchant and press the 'M' key to initiate buying and selling.

The scene transition is triggered by collision detection.

When the player reaches a specific point, they are transitioned to the new scene, such as an interior location or a new level.

This approach offers flexibility for future game design, allowing for seamless transitions between different areas.

Once inside the shop and interacting with the merchant, the player can buy and sell items.

Currently, I have implemented a sword for these transactions.

Over time, I plan to add more items, such as spells, armor, and other equipment.

I think my next step is to develop the main start menu.

After that is complete, I intend to merge the "merchant" branch with the main branch.

I'll then create a new, final branch, which I'm calling "menu," specifically for the game's main menu.

I don't think I will have time to develop much beyond that for now.

Today was a challenging yet rewarding day, filled with debugging, refining the system, and polishing up various aspects of my game development.

The main focus was on getting the buying and selling systems in the shopping UI to work properly.

It felt like a mini rollercoaster of breakthroughs and setbacks.

The initial issue was a frustrating problem with the inventory system and UI interaction.

After setting up the inventory slots and connecting signals for buying and selling items, clicking on items wasn't triggering any actions.

I discovered that the signal connections between the inventory slot UI and the actual logic behind buying and selling were misconfigured.

The signals were being emitted with incorrect arguments, causing errors in the on_buy_slot_clicked and on_selling_slot_clicked methods.

To fix this, I took extra care in verifying the parameters that the signals were sending.

The InventorySlot's signal slot_clicked needed to be bound correctly to pass the item index.

Using .bind(i) in the signal connection was key to solving this issue.

I also noticed some issues in the logic where clicking an item wouldn't update the UI correctly, and the buttons for buying and selling were disabled or not triggering.

I had to modify the button states to dynamically enable or disable them based on selected items.

This was especially important for the buy and sell buttons to function as intended.

I also encountered a tricky issue with the player's gold coin system.

When trying to print a message showing the item being bought, I mistakenly tried to access item_name, which didn't exist in the InventoryItem class.

Instead of item_name, I needed to use name, which was the correct property in my InventoryItem class.

This required a simple code fix in the print statement.

By the end of the day, I managed to resolve all the errors, and the buying/selling mechanics finally clicked into place.

The signals now work correctly, and the player's inventory reacts as expected when items are bought or sold.

The UI also correctly reflects the changes in the player's gold, with proper updates to the UI grids for buying and selling.

The experience was a bit overwhelming at times, but it was a great reminder of the importance of attention to detail in game development.

It feels rewarding to get this part of the game working, and now I can move on to the next challenge, maybe polishing up the item transaction logic or adding more intricate interactions.

I'm looking forward to more refinements and testing tomorrow!

Appendix B – Original timeline to the project

Term 1: Implementation Phase

- **Week 1-2:** Preparing the project plan, reading on design patterns, APIs, and graphics.
- **Week 3:** Study game development frameworks (e.g., Godot) and familiarize with core concepts, including 2D/3D rendering, physics, and scripting in GDScript.

- **Week 4-5:** Define the game concept and mechanics. Create a game design document (GDD) that outlines core gameplay elements, rules, and objectives

- **Week 5:** Implement the basic game engine architecture, focusing on setting up the scene, player movement, and core mechanics like physics, collision detection, and input handling.

- **Week 6:** Develop the initial game prototype, including key gameplay features (e.g., character control, environment interactions). Begin asset integration for basic visuals and sounds.

- **Week 7:** Research and integrate design patterns for game mechanics, such as agent behaviour and state machines for NPCs (non-playable characters) and enemies.

- **Week 8-9:** Fine-tune game mechanics and implement more advanced features, such as AI for enemies or puzzle logic. Begin testing for bugs and performance issues.

- **Week 10-11:** Prepare for the interim report and presentation, documenting the game's core features, progress, challenges, and next steps. Include the initial playable prototype for feedback.

Term 2: Refinement and Evaluation Phase

- **Week 1-2:** Refine and augment the game mechanics based on feedback. Re-evaluate and improve features like physics interactions, AI behaviour, or player progression systems.

- **Week 3:** Implement UI/UX elements, such as menus, scoreboards, and game controls. Create polished visual effects and animations to enhance user experience.

- **Week 4-5:** Encapsulate the game into a more modular structure using proper software engineering practices, ensuring the codebase is scalable and maintainable.

- **Week 6:** Integrate additional functionalities, such as multiplayer support, online leader boards, or game saving/loading mechanics.

- **Week 7-9:** Conduct extensive playtesting and debugging. Collect feedback from peers and make necessary adjustments. Optimize performance, polish visuals, and ensure compatibility across platforms.

- **Week 10-11:** Prepare for the final report and viva, presenting the completed game. Include evaluations on performance, user feedback, challenges encountered, and future improvements

Appendix C – Structure of the Submission

The project folder contains two primary subfolders: "***documents***" and "***product***," along with a "***diary.md***" file and other miscellaneous files. Notably, the "***diary.md***" file is located at the root level, outside of both the "***documents***" and "product" folders.

Within the "***documents***" folder, you will find the following files: the game design document, the interim report, the project plan, and the final report (provided for your reference, particularly for submission purposes).

The "***product***" folder encompasses several subfolders: "***assets***," "***resources***," "***scenes***," "***scripts***," and "***titlesets***." The "***assets***" folder contains all imported project assets, such as textures, swords, and tile sets. The "***resources***" folder holds files utilized by the coded engine. The "***scenes***" folder contains individual scenes dedicated to various game mechanics. Crucially, the "***scripts***" folder houses the code files, allowing for inspection of the project's logic. Lastly, the "***titlesets***" folder contains a file specific to the Godot engine.

Appendix D – Link to Project

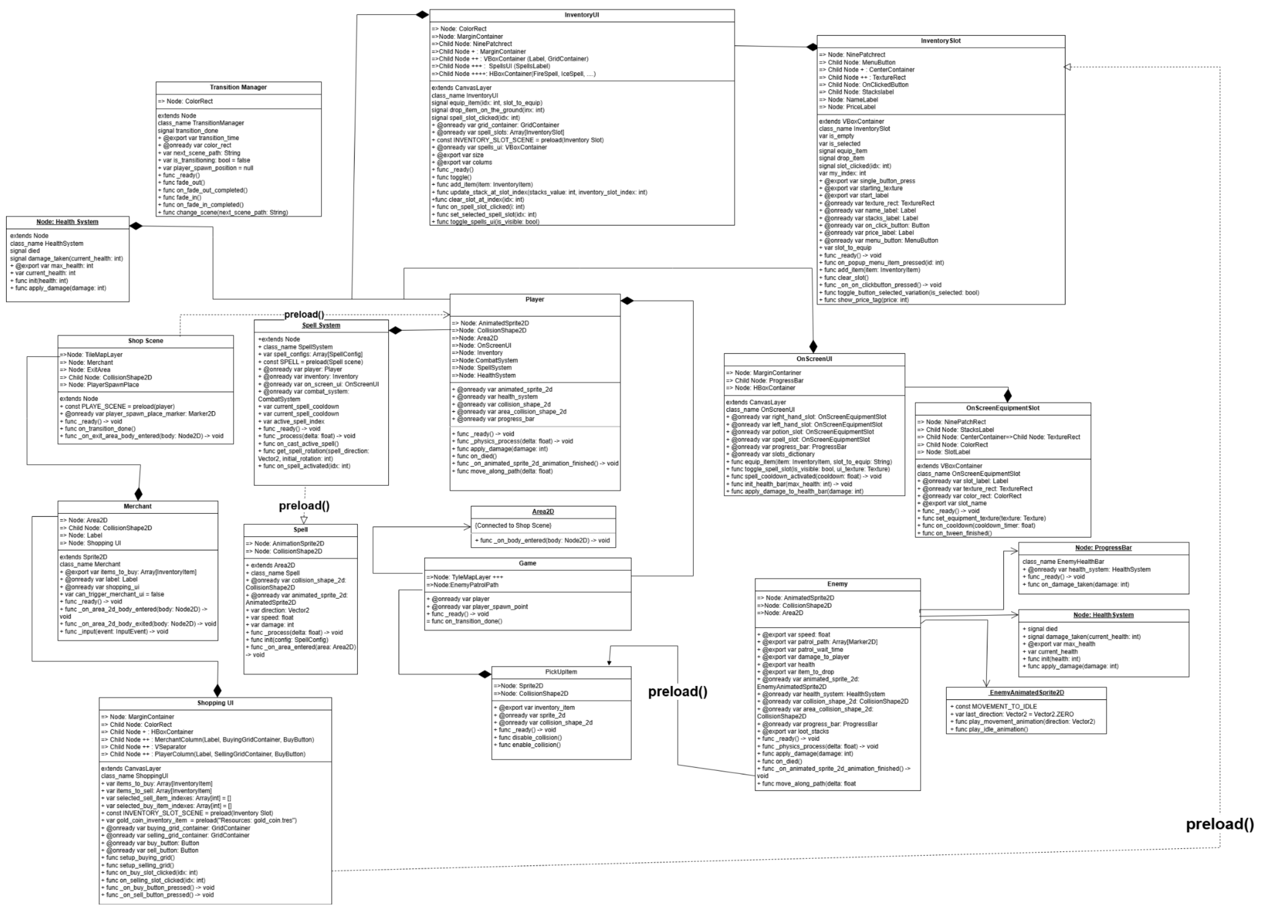
https://gitlab.cim.rhul.ac.uk/wlis107/PROJECT.git

Appendix E – Link to Video

https://www.youtube.com/watch?v=ysGUAJa71o0

Appendix F – Class Diagram

Mohammad Ibrahim Salman
101029175



```
InventoryUI
=> Node: ColorRect
=>Node: MarginContainer
=>Child Node: NinePatchrect
=>Child Node + : MarginContainer
=>Child Node ++ : VBoxContainer (Label, GridContainer)
=>Child Node +++ :  SpellsUI (SpellsLabel)
=>Child Node ++++: HBoxContainer(FireSpell, IceSpell, ....)
extends CanvasLayer
class_name InventoryUI
signal equip_item(idx: int, slot_to_equip)
signal drop_item_on_the_ground(inx: int)
signal spell_slot_clicked(idx: int)
+ @onready var grid_container: GridContainer
+ @onready var spell_slots: Array[InventorySlot]
+ const INVENTORY_SLOT_SCENE = preload(Inventory Slot)
+ @onready var spells_ui: VBoxContainer
+ @export var size
+ @export var colums
+ func _ready()
+ func toggle()
+ func add_item(item: InventoryItem)
+ func update_stack_at_slot_index(stacks_value: int, inventory_slot_index: int)
+func clear_slot_at_index(idx: int)
+ func on_spell_slot_clicked(i: int)
+ func set_selected_spell_slot(idx: int)
+ func toggle_spells_ui(is_visible: bool)
```

```
InventorySlot
=> Node: NinePatchrect
=> Child Node: MenuButton
=> Child Node + : CenterContainer
=> Child Node ++ : TextureRect
=> Child Node: OnClickedButton
=> Child Node: Stackslabel
=> Node: NameLabel
=> Node: PriceLabel
extends VBoxContainer
class_name InventorySlot
var is_empty
var is_selected
signal equip_item
signal drop_item
signal slot_clicked(idx: int)
var my_index: int
+ @export var single_button_press
+ @export var starting_texture
+ @export var start_label
+ @onready var texture_rect: TextureRect
+ @onready var name_label: Label
+ @onready var stacks_label: Label
+ @onready var on_click_button: Button
+ @onready var price_label: Label
+ @onready var menu_button: MenuButton
+ var slot_to_equip
+ func _ready() -> void
+ func on_popup_menu_item_pressed(id: int)
+ func add_item(item: InventoryItem)
+ func clear_slot()
+ func _on_on_clickbutton_pressed() -> void
+ func toggle_button_selected_variation(is_selected: bool)
+ func show_price_tag(price: int)
```

```
Transition Manager
=> Node: ColorRect
extends Node
class_name TransitionManager
signal transition_done
+ @export var transition_time
+ @onready var color_rect
+ var next_scene_path: String
+ var is_transitioning: bool = false
+ var player_spawn_position = null
+ func _ready()
+ func fade_out()
+ func on_fade_out_completed()
+ func fade_in()
+ func on_fade_in_completed()
+ func change_scene(next_scene_path: String)
```

```
Node: Health System
extends Node
class_name HealthSystem
signal died
signal damage_taken(current_health: int)
+ @export var max_health: int
+ var current_health: int
+ func init(health: int)
+ func apply_damage(damage: int)
```

```
Shop Scene
=>Node: TileMapLayer
=> Node: Merchant
=> Node: ExitArea
=> Child Node: CollisionShape2D
=> Node: PlayerSpawnPlace
extends Node
+ const PLAYE_SCENE = preload(player)
+ @onready var player_spawn_place_marker: Marker2D
+ func _ready() -> void
+ func on_transition_done()
+ func _on_exit_area_body_entered(body: Node2D) -> void
```

```
Spell System
+extends Node
+ class_name SpellSystem
+ var spell_configs: Array[SpellConfig]
+ const SPELL = preload(Spell scene)
+ @onready var player: Player
+ @onready var inventory: Inventory
+ @onready var on_screen_ui: OnScreenUI
+ @onready var combat_system:
CombatSystem
+ var current_spell_cooldown
+ var current_spell_cooldown
+ var active_spell_index
+ func _ready() -> void
+ func _process(delta: float) -> void
+ func on_cast_active_spell()
+ func get_spell_rotation(spell_direction:
Vector2, initial_rotation: int)
+ func on_spell_activated(idx: int)
```

```
Player
=> Node: AnimatedSprite2D
=>Node: CollisionShape2D
=>Node: Area2D
=>Node: OnScreenUI
=>Node: Inventory
=>Node:CombatSystem
=>Node: SpellSystem
=>Node: HealthSystem
+ @onready var animated_sprite_2d
+ @onready var health_system
+ @onready var collision_shape_2d
+ @onready var area_collision_shape_2d
+ @onready var progress_bar
+ func _ready() -> void
+ func _physics_process(delta: float) -> void
+ func apply_damage(damage: int)
+ func on_died()
+ func _on_animated_sprite_2d_animation_finished() -> void
+ func move_along_path(delta: float)
```

```
OnScreenUI
=> Node: MarginContariner
=> Child Node: ProgressBar
=> Node: HBoxContainer
extends CanvasLayer
class_name OnScreenUI
+ @onready var right_hand_slot: OnScreenEquipmentSlot
+ @onready var left_hand_slot: OnScreenEquipmentSlot
+ @onready var potion_slot: OnScreenEquipmentSlot
+ @onready var spell_slot: OnScreenEquipmentSlot
+ @onready var progress_bar: ProgressBar
+ @onready var slots_dictionary
+ func equip_item(item: InventoryItem, slot_to_equip: String)
+ func toggle_spell_slot(is_visible: bool, ui_texture: Texture)
+ func spell_cooldown_activated(cooldown: float) -> void
+ func init_health_bar(max_health: int) -> void
+ func apply_damage_to_health_bar(damage: int)
```

```
OnScreenEquipmentSlot
=> Node: NinePatchRect
=> Child Node: StacksLabel
=> Child Node: CenterContainer=>Child Node: TextureRect
=> Child Node: ColorRect
=> Node: SlotLabel
extends VBoxContainer
class_name OnScreenEquipmentSlot
+ @onready var slot_label: Label
+ @onready var texture_rect: TextureRect
+ @onready var color_rect: ColorRect
+ @export var slot_name
+ func _ready() -> void
+ func set_equipment_texture(texture: Texture)
+ func on_cooldown(cooldown_timer: float)
+ func on_tween_finished()
```

```
Merchant
=> Node: Area2D
=> Child Node: CollisionShape2D
=> Node: Label
=> Node: Shopping UI
extends Sprite2D
class_name Merchant
+ @export var items_to_buy: Array[InventoryItem]
+ @onready var label: Label
+ @onready var shopping_ui
+ var can_trigger_merchant_ui = false
+ func _ready() -> void
+ func _on_area_2d_body_entered(body: Node2D) ->
void
+ func _on_area_2d_body_exited(body: Node2D) -> void
+ func _input(event: InputEvent) -> void
```

```
Spell
=> Node: AnimationSprite2D
=> Node: CollisionShape2D
+ extends Area2D
+ class_name Spell
+ @onready var collision_shape_2d:
CollisionShape2D
+ @onready var animated_sprite_2d:
AnimatedSprite2D
+ var direction: Vector2
+ var speed: float
+ var damage: int
+ func _process(delta: float) -> void
+ func init(config: SpellConfig)
+ func _on_area_entered(area: Area2D)
-> void
```

```
Area2D
{Connected to Shop Scene}
+ func _on_body_entered(body: Node2D) -> void
```

```
Game
=>Node: TyleMapLayer +++
=>Node:EnemyPatrolPath
+ @onready var player
+ @onready var player_spawn_point
+ func _ready() -> void
= func on_transition_done()
```

```
PickUpItem
=>Node: Sprite2D
=>Node: CollisionShape2D
+ @export var inventory_item
+ @onready var sprite_2d
+ @onready var collision_shape_2d
+ func _ready() -> void
+ func disable_collision()
+ func enable_collision()
```

```
Enemy
=> Node: AnimatedSprite2D
=>Node: CollisionShape2D
=>Node: Area2D
+ @export var speed: float
+ @export var patrol_path: Array[Marker2D]
+ @export var patrol_wait_time
+ @export var damage_to_player
+ @export var health
+ @export var item_to_drop
+ @onready var animated_sprite_2d:
EnemyAnimatedSprite2D
+ @onready var health_system: HealthSystem
+ @onready var collision_shape_2d: CollisionShape2D
+ @onready var area_collision_shape_2d:
CollisionShape2D
+ @onready var progress_bar: ProgressBar
+ @export var loot_stacks
+ func _ready() -> void
+ func _physics_process(delta: float) -> void
+ func apply_damage(damage: int)
+ func on_died()
+ func _on_animated_sprite_2d_animation_finished() ->
void
+ func move_along_path(delta: float
```

```
Node: ProgressBar
class_name EnemyHealthBar
+ @onready var health_system: HealthSystem
+ func _ready() -> void
+ func on_damage_taken(damage: int)
```

```
Node: HealthSystem
+ signal died
+ signal damage_taken(current_health: int)
+ @export var max_health
+ var current_health
+ func init(health: int)
+ func apply_damage(damage: int)
```

```
EnemyAnimatedSprite2D
+ const MOVEMENT_TO_IDLE
+ var last_direction: Vector2 = Vector2.ZERO
+ func play_movement_animation(direction: Vector2)
+ func play_idle_animation()
```

```
Shopping UI
=> Node: MarginContainer
=> Child Node: ColorRect
=> Child Node + : HBoxContainer
=> Child Node ++ : MerchantColumn(Label, BuyingGridContainer, BuyButton)
=> Child Node ++ : VSeparator
=> Child Node ++ : PlayerColumn(Label, SellingGridContainer, BuyButton)
extends CanvasLayer
class_name ShoppingUI
+ var items_to_buy: Array[InventoryItem]
+ var items_to_sell: Array[InventoryItem]
+ var selected_sell_item_indexes: Array[int] = []
+ var selected_buy_item_indexes: Array[int] = []
+ const INVENTORY_SLOT_SCENE = preload(Inventory Slot)
+ var gold_coin_inventory_item  = preload("Resources: gold_coin.tres")
+ @onready var buying_grid_container: GridContainer
+ @onready var selling_grid_container: GridContainer
+ @onready var buy_button: Button
+ @onready var sell_button: Button
+ func setup_buying_grid()
+ func setup_selling_grid()
+ func on_buy_slot_clicked(idx: int)
+ func on_selling_slot_clicked(idx: int)
+ func _on_buy_button_pressed() -> void
+ func _on_sell_button_pressed() -> void
```